Comparando estruturas de Banco de Dados: Conheça a ferramenta EMS DB Comparer

Clube Delphi

Ano 13 :: Edição 148 :: R$ 11,90

DevMedia

Gerenciamento automático de metadados
Um framework automatizado para atualizar Estruturas de BDs

Generics + MVC
Indo além do conceitual

CERTIFICAÇÕES DELPHI

Dicas e Guias de Estudo

Delphi e RIA
Interfaces ricas para WEB, utilizando Delphi

JSON
Serializando objetos em 3 camadas

ISSN 1517990-7

DevMedia

DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior portal para desenvolvedores da América Latina!

20 mil posts

430 mil cadastrados

10 milhões de page-views por mês

http://www.devmedia.com.br/

DevMedia

Sumário

Delphi — Curso

06 – Certificação Delphi
[*Fabrício Hissao Kawata*]

Delphi — Banco de Dados

13 – Comparando estruturas de Banco de Dados
[*Paulo Quicoli*]

Mentoring — Delphi

19 – Gerenciamento automático de metadados
[*Paulo Quicoli*]

Boas Práticas — Boas Práticas

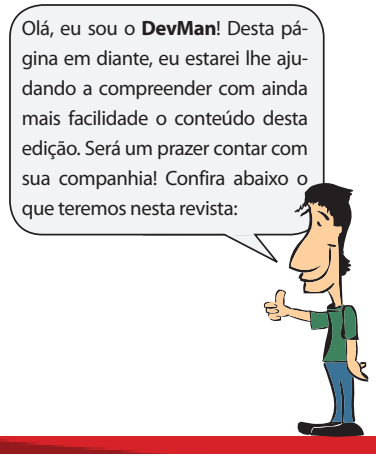
28 – JSON – Serializando Objetos
[*Filipe Dalepiane*]

Boas Práticas — Curso

37 – RIA com Delphi – Parte 2
[*Giuliano Scombatti Pinto*]

Boas Práticas — Delphi/Easy

43 – Generics + MVC = Boas práticas e reusabilidade
[*Ricardo Coelho*]



Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

RENOVE JÁ!

Sua assinatura pode estar acabando

Renovando a assinatura de sua revista favorita você ganha brindes e descontos exclusivos.

www.devmedia.com.br/renovacao ou (21)3382-5038

DevMedia

Clube Delphi

Ano 13 - 148ª Edição 2013 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral

Paulo Quicoli
pauloquicoli@gmail.com

Editor Técnico

Daniel Sobrinho Laporte
daniel.laporte@gmail.com

Equipe Editorial

Guinther Pauli
guintherpauli@gmail.com

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br

Giuliano Scombatti Pinto
giuliano@sygnux.com.br

Jornalista Responsável

Kaline Dolabella - JP24185

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

EDITORIAL

O assunto certificação técnica sempre causa barulho em meio à comunidade. Muitas empresas utilizam a certificação como um diferencial de nível salarial, outras até evitam profissionais certificados pensando que seu valor de trabalho é muito alto. Muitos desenvolvedores simplesmente ignoram, alegando que sua experiência é o que importa. Outros acreditam que a certificação é essencial para mostrar ao mercado sua capacitação. Mas, independente de tudo isso, a Embarcadero do Brasil traz de volta ao nosso mercado seu programa de certificações. Nessa edição temos um artigo do Fabrício que mostra todo o processo para obtenção do certificado. Além disso, ele apresenta os pontos que são mais requisitados nas provas e o que estudar. Em minha opinião, como o valor não é exorbitante, a prova seria uma ótima maneira de avaliar conhecimento, uma forma de medir o quanto se sabe da ferramenta.

Eu mesmo trago um artigo sobre um problema que não é visto como problema até sua empresa crescer. A atualização de estrutura de banco de dados. É mostrado um pequeno framework que se encarrega de aplicar instruções SQL de forma automática, sem intervenção humana, assim, diminuindo o tempo de uma atualização de versão de software. Contudo, conforme será visto, esse framework não realiza a comparação de base de dados, ele executa o script contendo a diferença. Para gerar essa diferença é preciso utilizar ferramentas específicas, como o EMS DB Comparer, também apresentado por mim em um artigo que podemos dizer é até complementar.

Continuando sobre interfaces ricas, o Giuliano mostra desta vez como personalizar ainda mais a exibição da aplicação, incluindo desta vez o próprio Google Maps na interface. Não perca.

Como é de praxe, temos uma seção dedicada para aqueles que estão iniciando com Delphi. Esta edição traz um artigo do Ricardo sobre Generics, mas este artigo vai mais além, ele demonstra seu emprego prático em um aplicativo Win32 que faz uso do padrão MVC. Acredito que mesmo que você não seja mais iniciante, precisa verificar a implementação sugerida, seja para aprender ou relembrar.

Por fim, o Filipe traz um artigo que servirá de referencial para muitos. Como aplicar JSON em uma aplicação três camadas que utiliza padrões de projeto e de injeção de dependência. Tudo isso em uma solução completa e funcional.

Que todos tenham uma boa leitura.

Paulo Quicoli
pauloquicoli@gmail.com
Editor Geral
quicoli.wordpress.com
twitter.com/pauloquicoli

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DevMedia

Seu **conhecimento vale muito** na DevMedia

Publique artigos e vídeos gratuitamente no site e participe do programa **PayContent.**

Sua participação pode ser premiada: quanto mais você postar, mais chances de ganhar.

Para saber mais acesse:
www.devmedia.com.br/paycontent

DevMedia

Certificação Delphi – Parte 1

Conheça o novo programa de certificação da Embarcadero

Este artigo faz parte de um curso

Resumo DevMan

Certificação Delphi:

Ainda na época da Borland, o Delphi já provia de um programa de certificação, que era a forma mais reconhecida e almejada do profissional demonstrar, a quem quer que seja, toda sua habilidade e conhecimento sobre a linguagem e a tecnologia em si. Com o passar do tempo e a troca de fabricante do produto (de Borland para CodeGear), o programa de certificação acabou ficando em segundo plano, até sair de cena por completo, o que causou muitos questionamentos e debates em meio à comunidade. Por conseguinte, com a nova mudança de direção, agora de CodeGear para Embarcadero, esta veio a dar uma sobrevida à Certificação Delphi, através de uma nova estratégia que envolvia todo um trabalho de reestruturação do programa. O resultado desta operação pode finalmente ser vista em meados de 2011, com o anúncio oficial do novo programa de certificação Delphi, que é o tema central deste artigo.

Em que situação o tema é útil:

Ser um profissional certificado não necessariamente lhe dá garantias de um bom emprego ou maiores rendimentos e salários, mas pode o diferenciar em meio a um agregado de desenvolvedores não certificados. Isto porque o processo de certificação avalia os conhecimentos do profissional sobre a programação Delphi de uma forma geral, da parte mais conceitual à parte mais prática. Em vista disso, o próprio mercado tende a valorizar mais um profissional com tal gabarito que, além de uma bagagem técnica, dá mostras de foco e perseverança em alcançar seus objetivos. Adicionalmente, a certificação é uma forma de satisfação do próprio ego do desenvolvedor, que vê nela uma forma de demonstrar e estabelecer suas habilidades.

O novo programa de certificação Delphi da Embarcadero, lançado recentemente, segmenta a certificação em dois tipos distintos, apresentados como Delphi Developer e Delphi Master Developer. Obviamente ambas são fundamentadas essencialmente sobre o contexto Delphi, tanto no aspecto de linguagem quanto da tecnologia. A diferença entre ambas fica por conta do nível de conhecimento exigido e, consequentemente, o seu grau de dificuldade. Um detalhe a ser mencionado já de início é que este novo programa é todo elaborado nas capacidades do Delphi até sua versão XE. Sendo assim, este é um fator a se levar em conta no momento da preparação, principalmente no que diz respeito aos aspectos do IDE e recursos da linguagem.

Certificação – Delphi Developer

A certificação Delphi Developer envolve o teste mais básico do programa e, portanto, o ponto de partida para o desenvolvedor. De uma forma geral, nele será avaliado o nível de seu conhecimento sobre os conceitos básicos e essenciais envolvendo a programação Delphi. Isso significa dizer que a ênfase da avaliação se dará sobre a sintaxe da linguagem, assimilação e utilização do ambiente de desenvolvimento (IDE), domínio das técnicas básicas de programação e desenvolvimento de software com banco de dados no Delphi.

Requisitos para a realização da prova

O requisito mínimo sugerido para a realização do exame desta certificação é o candidato possuir de um a dois anos de experiência em programação Delphi, que é o tempo estimado pela Embarcadero para que a pessoa adquira um nível de conhecimento essencial para a realização do exame. Este conhecimento deverá então ser complementado por um estudo mais direcionado dos temas envolvidos. Além disso, olhando para a comunidade brasileira, outro requisito que

deve ser levado em conta é a questão do idioma, uma vez que o conteúdo das provas é todo em inglês. Isso exige uma atenção maior por parte do candidato, tanto na preparação quanto na execução da mesma.

Preparação

A preparação para uma prova de certificação, seja qual for a tecnologia em questão, envolve essencialmente a revisão e o estudo dos tópicos apontados pela fabricante ou elaborada do exame. No caso da certificação Delphi, quem provê tais informações é a própria Embarcadero, através de um Guia de Estudos (Study Guide) direcionado. Em vista dos assuntos relacionados aos tópicos propostos, são fornecidas algumas referências para estudo, tanto online (Ex: links) como também de materiais didáticos (Ex: livros e revistas). Neste guia também estão contidos algumas dicas relacionadas à certificação, tal como uma seção de perguntas e respostas, para que o desenvolver se familiarize com o estilo da prova.

Na contramão desta abordagem autodidata estão os treinamentos oficiais providos pela Embarcadero, disponibilizados em vários parceiros espalhados pelo país, e que irão auxiliar o candidato em sua preparação. Até o momento da escrita do presente artigo, os cursos oficiais disponíveis e condizentes com a certificação Delphi Developer são: Delphi Essentials (ver **Nota do DevMan 1**) e Desenvolvimento de aplicações Win32 (ver **Nota do DevMan 2**).

Nota do DevMan 1

Delphi Essentials é o nome do treinamento oficial da Embarcadero voltado aos profissionais que ainda não utilizam o Delphi e desejam conhecer a ferramenta. Sendo assim, ele é tido como um treinamento "inicial". Seu conteúdo programático inclui: overview da ferramenta, Delphi Language, tipos de aplicações, introdução ao desenvolvimento com banco de dados e modelagem UML.

Nota do DevMan 2

Desenvolvimento de Aplicações Win32 é o nome do treinamento oficial da Embarcadero voltado aos profissionais que já possuem conhecimento em Delphi, cujo foco é o desenvolvimento de aplicações com banco de dados. Ele funciona como uma espécie de "etapa seguinte" ao treinamento Delphi Essentials. Seu conteúdo programático inclui, entre outras coisas: estudo do IDE, OO, Regras de negócio, UML, Design Patterns, elementos de UI, tratamento de exceções, depuração, pacotes, dbExpress, TClientDataSet, gráficos, relatórios e desenvolvimento avançado com banco de dados.

Custo da prova

Para realizar a prova o candidato deve adquirir um código (uma espécie de código do produto) junto à Embarcadero, sob o custo de U$49 dólares. Uma vez adquirido e registrado, o código pode ser utilizado em até 90 dias. Após este período ele é expirado, e um novo código deve ser adquirido para a realização do exame.

Realização da prova

Com o código da prova em mãos, o candidato já está politicamente apto à realização da prova, que acontece de forma totalmente online. Isto quer dizer que a partir de qualquer computador com acesso à internet, basta que o desenvolver acesse um site específico da Embarcadero (ver seção Links) para ativar o seu código. Por conseguinte, uma conta é instantaneamente criada para que você tenha acesso ao exame. Da mesma forma que a realização da prova, o seu resultado também é online. Sendo assim, um momento após realizar o teste você já recebe sua nota, bem como um resumo de erros e acertos para cada tópico abordado.

A prova de certificação Delphi Developer é composta de um mix de 60 questões de múltipla escolha, todas de mesmo peso, em inglês, provindas de 11 grupos de assuntos distintos:

- Princípios fundamentais do Delphi;
- Tipos de dados, variáveis e constantes;
- Procedimentos e Funções;
- Classes e Objetos;
- Rotinas Padrão e I/O;
- Bibliotecas e Pacotes;
- Interfaces de Objeto;
- Gerenciamento de Memória;
- Generics, Atributos e Métodos Anônimos;
- Exceptions e Assertions;
- Desenvolvimento e conceitos de banco de dados.

A partir de cada grupo, um número aleatório de perguntas é selecionado, com a obrigatoriedade de que ao menos uma pergunta de cada grupo faça parte do exame O tempo limite para a realização do exame é de 60 minutos, o que resulta numa média de 1 minuto para a resolução de cada questão.

A aprovação

O percentual de acertos que o candidato necessita atingir para que seja aprovado no exame é de 80%, o que resulta em 48 questões corretas. Novamente olhando em termos de média, podemos estabelecer aproximadamente 1 erro para cada grupo de assunto.

Uma vez aprovado, o desenvolvedor se torna um membro especial, reconhecido pela própria Embarcadero em meio à comunidade global Delphi, sob o status de Certified Delphi Developer. Para ratificar esta bagatela, o candidato recebe um documento digital de certificação, nomeado, no formato PDF. Além disso, neste momento você também se torna apto a realizar o exame para a certificação Delphi Master Developer. Diz-se apto não pelo fato da obrigatoriedade condicional em se ter a certificação Delphi Developer para poder realizar o exame para Delphi Master Developer, mas sim em função do caminho natural de evolução do seu conhecimento.

A certificação Delphi Developer tem a validade de dois anos, a contar a partir da data da realização do exame.

Tópicos de estudo

A lista de tópicos de estudo envolvida na certificação Delphi Developer mostra-se bastante extensa num primeiro momento.

No entanto, à medida que o candidato se aprofunda nas áreas exigidas, descobre que a maioria dos conceitos envolvidos está fortemente presente em seu cotidiano profissional. O ponto chave aqui é o entendimento e assimilação dos detalhes relacionados a cada assunto, que podem ser os diferenciais numa avaliação. A seguir é feito um overview de cada tópico envolvido baseado, como não poderia deixar de ser, no roteiro do Study Guide.

Fundamentos Delphi

Fundamentos Delphi é tópico que engloba o conhecimento do produto em duas vertentes: linguagem e ferramenta. Em termos de linguagem, a Delphi Language é uma linguagem compilada, baseada em Object Pascal, e considerada de alto nível. É dito também que ela é fortemente tipada, pelo fato de dispor de uma grande variedade de tipos de dados singulares e insubstituíveis, o que contribui para a inteligência do compilador. Através de seus inúmeros recursos, provê apoio suficiente tanto ao desenvolvimento estruturado quanto ao desenvolvimento orientado a objetos.

O Delphi, como ambiente integrado de desenvolvimento (IDE) contribui para o desenvolvimento rápido de aplicações, sendo assim considerada uma ferramenta RAD (Rapid Application Development). Uma característica marcante desta afirmação é o seu conhecido poderio de arrastar-e-soltar componentes na construção de aplicações, dos mais variados segmentos.

No que diz respeito à doutrina da Delphi Language, será necessário ter o conhecimento de sua sintaxe, suas palavras reservadas e suas nuances relacionadas à declaração de variáveis, procedimentos (procedures) e funções (functions). Já no aspecto do produto como ferramenta de desenvolvimento, é fundamental o conhecimento sobre as capacidades essenciais do Delphi XE e os tipos de arquivos envolvidos no contexto de um projeto Delphi.

Tipos de dados, variáveis e constantes

No Delphi, um tipo nada mais é do que uma nomenclatura dada a um determinado tipo de dado. A linguagem disponibiliza então vários tipos de dados, desde tipos simples como String e Integer a tipos estruturados, como Record e Array.

De forma conjunta aos tipos de dados, está o emprego de variáveis e constantes. Uma variável é um nome dado a um local reservado na memória e cujo valor pode ser alterado em tempo de execução do programa. Além disso, ela pode ser classificada em dois tipos: global e local. Uma variável, quando definida como local, somente fica disponível ao escopo de sua declaração, enquanto que quando definida como global, fica acessível para o contexto da aplicação.

Ao contrário das variáveis, as constantes no Delphi são identificadores cujo valor não pode ser alterado em tempo de execução. Por este motivo, o próprio compilador é capaz de identificar seu tipo, a partir de seu valor definido. Num exemplo primário, ao se declarar uma constante com o valor 'clubedelphi', o compilador já faz a identificação de seu tipo implícito como sendo String. Adicionalmente, é possível ainda fazer a declarar explícita de seu tipo, tal como a seguir:

```
const
  MinhaConstante: string = 'clubedelphi';
```

Neste tópico o conhecimento do candidato será testado com relação a três aspectos condizentes à linguagem: tipos de dados, variáveis e constantes. Para todos eles, o fundamental é saber os detalhes de sua sintaxe de declaração, suas diferenças e nuances técnicas de uso. Exclusivamente para a parte de Data Types, é importante assegurar um claro entendimento, de forma geral, sobre os tipos de dados disponíveis no Delphi. Isto inclui ponteiros, records, enumerations, matrizes estáticas e dinâmicas (arrays), bem como as vertentes do tipo string e char, tais como Char, String, AnsiChar, AnsiString, WideChar e UnicodeString. Exige-se também o conhecimento das funções RTL (Run Time Library) para manipulação de conjuntos de caracteres.

Procedimentos e funções

Já habituais no cotidiano de todo programador Delphi, os métodos, definidos na forma de procedures e functions durante o

desenvolvimento, também são pontos abordados nesta certificação. Conceitualmente falando, tais procedimentos e funções são a representação das diversas rotinas utilizadas por um programa. Elas são caracterizadas então como sendo blocos independentes de instruções de código que podem ser chamados a partir de diferentes pontos da aplicação. A grande diferença entre elas se dá pelo fato de que uma Procedure, quando chamada, não retorna valor, enquanto que uma Function retorna.

O nível de conhecimento exigido para Procedures e Functions passa por sua sintaxe de declaração, suas formas e convenções de chamada, aninhamentos (procedures e funções aninhadas) até a sobrecarga de métodos (overloading). Além disso, outros pontos relacionados também são considerados, tais como parâmetros e valores de retorno (no caso de funções), diretivas e formas de declarações (global e local).

Classes e Objetos

Outro tópico exigido na prova de certificação Delphi Developer diz respeito à programação orientada a objetos, mais especificamente a classes e objetos. Uma classe nada mais é do que uma abstração para um grupo de objetos com características similares. Em sua estrutura, basicamente são definidos os comportamentos destes objetos (métodos), bem como os atributos pertinentes. Já um objeto é a instância de uma classe. Trançando um paralelo com a vida real, podemos determinar que uma classe é como se fosse a planta de uma casa, enquanto que o objeto seria a casa propriamente dita. Este talvez seja um dos exemplos mais tradicionais e utilizados no meio acadêmico, para se explicar a relação existente entre classes e objetos.

Olhando pelo lado prático, todo desenvolvedor Delphi de média data tem pleno conhecimento de que mesmo que não se faça uso pleno da orientação a objetos no desenvolvimento Delphi, a nuances da OO estão mais do que presentes em todo o contexto ao qual estamos acostumados a trabalhar. Prova disso são as utilizações de herança, principalmente de formulários, construtores e destrutores, polimorfismo e visibilidade de membros de classes e objetos – public, private, protected, published, entre outros. Temas como estes são os exigidos nesta certificação.

Rotinas padrão e I/O

O tema "Rotinas padrão" exige um conhecimento sobre o uso da Run Time Library (ou simplesmente RTL), que é biblioteca de tempo de execução do Delphi. Sendo assim, é desejável que se tenha o entendimento adequado das funções RTL disponíveis, incluindo a manipulação de strings e alocação de memória. Na mesma medida, "I/O" requer o conhecimento sobre as rotinas de entrada (Input) e saída (Output), tais como o uso dos métodos padrões Read/ReadLn e Write/WriteLn, tratamento de erros e manipulação de arquivos e diretórios.

Bibliotecas e Pacotes

Bibliotecas e Pacotes são tratados no meio do desenvolvimento Delphi como DLLs (Dynamic Link Libraries) e Packages, respectivamente. Uma DLL pode ser definida como sendo uma biblioteca de vínculo dinâmico, cuja representação é um arquivo compilado de extensão .dll. Uma vez construída, uma DLL pode ser chamada a partir de qualquer tipo de aplicação, seja esta escrita em Delphi ou não. Sua utilização envolve desde a centralização de rotinas à modularização de aplicações, uma vez que sua estrutura pode conter Units e Forms.

De forma semelhante, as Packages se apresentam como um tipo especial de DLL, exclusivas ao cenário Delphi. São representadas por arquivos .bpl (acrônimo de Borland Package Library). Sua estrutura, assim como o das DLLs, também suportam Units e Forms, num processo de modularização da aplicação. As Packages podem ser de dois tipos, Design Time e Runtime, e sua utilização inclui desde a instalação de componentes no IDE ao envolvimento com aplicações.

Para as Libraries, o nível de exigência para a certificação envolve basicamente a criação e utilização de rotinas em DLLs para Windows. Já para as Packages, deve-se ter o entendimento sobre os fundamentos básicos de um pacote, que envolve desde sua utilização, convenção de nomenclatura e tipos (Design Time e Runtime), até a compreensão de suas cláusulas contains e required.

Interfaces

Aqui, o termo Interfaces diz respeito a interfaces de objetos e não a interfaces de usuário (UI). Uma Interface estabelece uma espécie de "contrato" que deverá ser seguido pela estrutura que declara sua implementação. Em outras palavras, uma classe que implemente uma determinada Interface deverá, obrigatoriamente, seguir o contrato especificado por esta. Neste contrato, pode estar definido, por exemplo, a assinatura de um método, que deverá então ser implementado pela classe.

Na certificação Delphi Developer, o que será avaliado será o seu conhecimento sobre o uso de interfaces na programação com a linguagem Delphi. Isto inclui a sintaxe padrão de declaração de uma Interface no Delphi, suas formas de implementação, o seu uso com classes, bem como as diferenças e ganhos existentes em abordagens que façam uso de Interfaces ao invés da herança.

Gerenciamento de memória

No mundo da programação, o termo Memory Management faz alusão aos recursos de gerenciamento de memória providos por determinada linguagem. Como não poderia ser diferente, o Delphi possui seu próprio gerenciador de memória e, consequentemente, uma gama de recursos para a realização do processo. Dessa forma, a exigência aqui é pelo conhecimento sobre a utilização das funções de gerenciamento de memória presentes no Delphi, os tipos de gerenciamento (automático e manual), bem como a configuração do próprio gerenciador. Adicionalmente, alguns outros pontos também relacionados ao tema são pautados, tais como tratamentos de vazamentos de memória (memory leaks), envolvimento das áreas de memória (Stack e Heap) e as nuances da alocação de memória para tipos de dados, classes, objetos, matrizes e strings.

Generics, Attributes e Métodos Anônimos

De forma sucinta, Generics podem ser definidos como sendo estruturas parametrizadas por um tipo. Num exemplo clássico, temos possibilidade de definição de uma lista de objetos de um tipo não definido (genérico), a ser conhecido no momento de sua utilização: List<T>, onde T é um tipo qualquer a ser informado no momento do seu uso. No Delphi, o termo Generics faz referência basicamente a tipos, estruturas e métodos.

Atributos, ou Attributes, são um recurso da linguagem Delphi que permite se fazer "anotações" com objetos especiais em um tipo (ex: uma classe) e em seus membros (ex: um campo da classe). Tais objetos são então capazes de carregar alguma informação adicional a ser recuperada em runtime, visando os mais variados fins. Um exemplo comum do uso de Attributes é na construção de frameworks.

Conforme seu próprio nome sugere, o Método Anônimo (ou Anonymous Method) é um método (procedure ou function) sem um nome associado, cujo bloco de código é tratado como um elemento que poderá ser utilizado como um parâmetro para um método ou mesmo atribuído a uma variável. A estrutura básica de um método anônimo pode ser visto na **Listagem 1**. Na **Listagem 2**, é mostrado um exemplo sucinto de uso de Anonymous Method no Delphi.

Listagem 1. Estruturas básicas para métodos anônimos

```
// Método anônimo - Procedure
procedure (parâmetros)
begin
 // código
end;

// Método anônimo - Function
function (parâmetros): tipo de retorno
begin
 // código
end;
```

Listagem 2. Estruturas básicas para métodos anônimos

```
type
 // declarando um tipo Anonymous Method
 TMyFunction = reference to function(N1, N2: Integer): Integer;
...

var
 MyFunc: TMyFunction; // declarando uma variável do tipo TMyFunction
begin
 // método anônimo sendo atribuído à variável
 MyFunc := function(N1, N2: Integer): Integer
           begin
             Result := N1 + N2;
           end;
 ...
end;
```

Por se tratarem de temas até certo ponto "novos" no Delphi, a exigência de conhecimento sobre Generics, Attributes e Anonymous Methods é bastante básica, ou seja, o candidato deve estar alinhado quanto à sintaxe de declaração de cada uma e suas utilizações mais comuns. Indo um pouco além, o guia de estudo ainda pontua o uso da RTTI (Run Time Type Information) extendida, generic constraints e os tipos de classes de coleção.

Exceptions e Assertions

No Delphi, uma exceção (Exception) é uma ação levantada no momento em que ocorre um erro na execução do programa e que acaba por interromper seu fluxo normal. O controle do fluxo é então transferido a um manipulador de exceção, que permitirá o tratamento do erro. Quando um erro efetivamente encerra um aplicativo ele é classificado como um erro de tempo de execução, ou runtime error. Um dos erros mais comuns deste tipo é o indicativo de memória insuficiente (insufficient memory). Para se fazer a devida captura e tratamento de erros deste tipo, convertendo-os em simples exceções, basta que a aplicação ou unit em questão faça uso da unit SysUtils do Delphi.

No contexto da programação, uma exceção nada mais é do que um objeto, ou seja, a instância de uma classe. Isso significa dizer que o Delphi possui várias classes de Exceptions, cada qual com sua finalidade e direcionamento como, por exemplo, EMathError, EZeroDivide e EDatabaseError.

Assertions podem ser definidos basicamente como sendo procedimentos baseados em uma condição e uma mensagem de erro opcional. Sendo assim, são indicados para testes de condições não esperadas para o programa. Na prática, quando uma Assert falha o fluxo de execução do programa também é interrompido por um erro de tempo de execução ou, opcionalmente, é levantada uma exceção do tipo EAssertionFailed.

Neste tópico, é desejável que você saiba como utilizar e declarar Exceptions e Assertions na linguagem Delphi; como criar manipuladores de exceção; ter um conhecimento sobre as Classes de exceção disponíveis no Delphi; como trabalhar com Exceptions aninhadas; como fazer o raise de exceções; uso do método Abort; uso de exceções silenciosas; ativação e desativação de Assertions.

Desenvolvimento Delphi com banco de dados

O desenvolvimento com banco de dados já é uma tradição no Delphi e, portanto, tende a ser algo familiar à grande maioria dos desenvolvedores. Sendo assim, o desejado aqui é que o candidato tenha um conhecimento geral sobre os conceitos envolvidos a esta prática, tais como:

- Uso dos componentes de acesso a dados, tais como os do grupo dbExpress e as combinações e configurações necessárias para a utilização dos mesmos em uma aplicação;
- A forma e a sequência dos métodos e eventos disponibilizados por estes componentes;
- Uso dos componentes data-aware, os famosos componentes "DB" presentes no Delphi, tais como TDBEdit, TDBGrid, TDBListBox e TDBCheckBox;
- A forma e os detalhes que envolvem as conexões com banco de dados a partir de uma aplicação Delphi.

• Drivers dbExpress disponíveis para diferentes bancos de dados;
• Manipulação de registros provindos de um banco de dados – inserção, edição, exclusão, atualização e navegação;
• Conhecimento das classes TDataSet e TField, que fazem parte da essência do desenvolvimento com banco de dados no Delphi. TDataSet, por exemplo, é a classe ancestral de componentes de acesso a dados como TClientDataSet e TSQLDataSet, da mesma forma que TField é a classe base para seus campos, vide TStringField, TSmallintField e TIntegerField;

Perguntas e Respostas

Conforme dito anteriormente, o guia de estudos provido pela Embarcadero para a certificação Delphi Developer apresenta em seu conteúdo algumas "perguntas e respostas" nos moldes da prova. A intenção é expor ao desenvolvedor o estilo seguido pelo exame, tanto pelo lado do conteúdo exigido, quanto pelo lado da disposição das perguntas e alternativas de resposta. Para dar mostras disso, são apresentadas a seguir, de forma traduzida, algumas das "perguntas e respostas" disponibilizadas no guia. De forma a contribuir ainda mais para o pleno entendimento das mesmas, a cada resposta é feito um breve comentário de justificativa da mesma.

1. Na linguagem Delphi, que símbolos ou palavras definem o início e o fim de um bloco de código?

a) {}
b) []
c) start/end
d) begin/end

Resposta: D.
Justificativa: As palavras begin e end demarcam, respectivamente, o início e o fim de um bloco de código na linguagem Delphi (**Figura 1**).

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // Código
end;
```

Figura 1. Exemplo de bloco de código no Delphi

2. Qual é o tipo subjacente de TDateTime?

a) Double
b) TObject
c) String
d) Integer

Resposta: A.
Justificativa: O tipo subjacente, ou seja, o tipo básico de TDateTime é o Double. Isso pode ser comprovado no próprio Delphi, na declaração de TDateTime na unit System ou pelo próprio code completion do IDE (**Figura 2**).

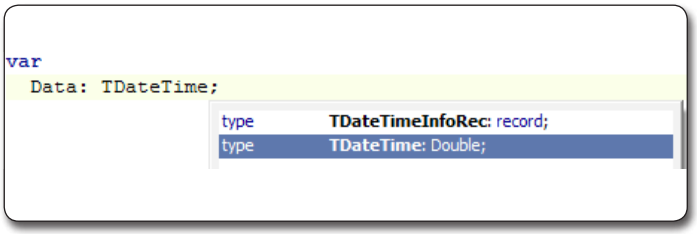


Figura 2. Tipo TDateTime

3. No Delphi, pacotes (Packages) podem conter componentes e classes não-visuais, mas não formulários (Forms).

a) True
b) False

Resposta: B.
Justificativa: A afirmativa proposta está incorreta, pois no Delphi, uma Package pode também conter formulários, sendo assim uma forma eficiente de se modularizar uma aplicação. A **Figura 3** mostra a estrutura de uma Package contendo um Form em sua estrutura.

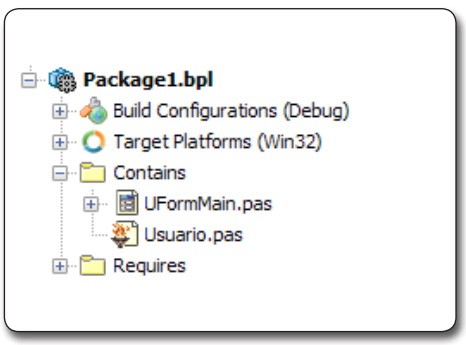


Figura 3. Estrutura de uma Package

4. Na linguagem Delphi, que símbolos ou palavras definem a declaração de um atributo (attribute)?

a) {}
b) <>
c) [ ]
d) &&
e) Nenhuma das opções

Resposta: C.
Justificativa: Definidos como sendo um recurso da linguagem Delphi, os atributos (Attributes) permitem anotações sobre tipos e seus membros, utilizando-se objetos contendo informações adicionais, que podem ser recuperadas em tempo de execução (runtime). A declaração de um atributo é definida por [], tal como:

```
TDemoObject = class
 private
   [MyDemo(TStringList)]
   FField: Integer;
...
end;
```

5. Em que área da memória são armazenadas as variáveis locais e parâmetros de Procedures?

a) Stack
b) Heap
c) Aleatório

Resposta: A.
Justificativa: Um programa Delphi em execução utiliza duas regiões de memória para armazenar algumas de suas informações: o Stack e o Heap. O Stack é o local da memória onde são então armazenados vários tipos de dados, tais como os valores de certos tipos de variáveis, bem como os parâmetros passados para métodos.

6. Ao se trabalhar com um grande número de registros em um DataSet, qual método do próprio DataSet deve ser chamado, a fim de se evitar a atualizações desnecessárias nos controles data-aware em tela?

a) BlockUpdate
b) DisableControls
c) Disable
d) Nenhuma das alternativas

Resposta: B.
Justificativa: DisableControls é um método declarado na classe TDataSet que desativa a exibição de informação em todos os controles data-aware ligados a um DataSet (tal com um TClientDataSet), otimizando processos onde esta atualização é desnecessária. Sua utilização comumente é combinada com o emprego do método EnableControls, que ativa a exibição nos controles. Uma situação clássica de uso do método DisableControls se dá em rotinas de iteração:

```
...
MyDataSet.DisableControls; // "desabilita" os controles
 try
  MyDataSet.First;
  while not MyDataSet.Eof do // inicia a iteração
  begin
   ShowMessage(MyDataSet.Fields[0].AsString);
   MyDataSet.Next;
 end;
 finally
  MyDataSet.EnableControls; // "habilita" os controles
 end;
...
```

Seguindo o molde desta apresentação, na próxima (e última) parte do artigo será abordada a certificação Delphi Master Developer e todos os detalhes envolvidos.

Conclusão

"A conquista de uma certificação pode ajudar o desenvolvedor a evoluir dentro da empresa ou abrir as portas para novas oportunidades. Ela pode ainda ser solicitada por empregadores potenciais como parte de um processo de contratação.". Fazendo uso das palavras da própria Embarcadero, este artigo teve como objetivo expor ao leitor o cenário atual das Certificações Delphi. De forma abrangente, pudemos ver as opções disponíveis, o nível de exigência para a certificação Delphi Developer, bem como os aspectos técnicos exigidos para estudo.

Portanto, se você é desenvolvedor Delphi, apresenta um nível satisfatório de conhecimento e deseja dar um passo a mais em sua trajetória profissional ou mesmo pessoal, a certificação é um caminho. Com certeza você se verá em um "teste de fogo", em que é colocada à prova toda sua habilidade e conhecimento adquirido ao longo de sua trajetória de programação. O resultado final disso é você ter seus esforços reconhecidos pela Embarcadero e pela própria comunidade de desenvolvedores Delphi espalhadas pelo globo. Mais um desafio está lançado. O usufruto dos benefícios proporcionados fica com você!

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ. Atua como Analista Programador Delphi há 8 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Blog: prismsilver.blogspot.com

Delphi XE3 – Download Trial
https://downloads.embarcadero.com/free/delphi

Delphi XE3 – Página oficial do produto
http://www.embarcadero.com/products/delphi

Embarcadero Delphi Developer Certification
http://www.embarcadero.com/br/certification/delphi-developer

Embarcadero Certification Code Activation
http://certification.embarcadero.com/reg

Atendimento - Embarcadero do Brasil
http://www.embarcadero.com/br/how-to-buy

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Comparando estruturas de banco de dados

Conheça o EMS DB Comparer

Quando existe um banco de dados em produção e outro em desenvolvimento, é muito provável que esse de desenvolvimento sofra alterações que deverão ser aplicadas na versão de produção. Manter essas diferenças pode ser um trabalho difícil, caso o desenvolvedor não esteja utilizando ferramentas específicas. Neste artigo conheceremos a ferramenta EMS DB Comparer. Criada pela empresa EMS Database Management Solutions e atualmente na versão 3.3.1.6 (até o fechamento deste artigo). Entre as principais encontram-se:

- Comparação entre bancos de dados ou armazenados em diferentes servidores;
- Comparação de todos os objetos de um banco de dados, ou somente os selecionados;
- Sincronização entre os bancos de dados comparados;
- Geração de relatórios com as diferenças dos bancos de dados;
- Habilidade de automatizar a comparação e sincronização utilizando a aplicação console disponível;
- Suporte às versões mais recentes do Interbase e Firebird;
- Editor de scripts SQL com sintax highlight;
- Criação de templates para facilitar/acelerar edição de scripts SQL;
- Permite a navegação sincronizada entre os objetos dos dois bancos de dados comparados;

Resumo DevMan

Comparando estruturas de banco de dados:

O produto DB Comparer realiza comparações entre bancos de dados InterBase/Firebird, mas não se limita somente a esses SGBDs. Oferece suporte também para SQL Server, Oracle, MySQL e PostgreSQL, possibilitando assim visualizar diferenças em estruturas e gerar scripts para eliminar diferenças nos principais bancos de dados do mercado. Sua comercialização se dá liberando uma versão específica para cada tipo de banco de dados e aqui vamos tratar da versão para Firebird e Interbase.

Em que situação o tema é útil:

Quando é preciso manter duas bases de dados diferentes, com a mesma estrutura. Essa situação é muito comum enquanto um software está em fase de desenvolvimento e parte dele já está em produção. Ao incluir novos recursos, pode ser necessário incluir novas tabelas, campos ou outros recursos. Ao atualizar o software, o banco de dados também precisa ser atualizado. É nisso que ferramentas de comparação auxiliam.

Tutorial

Instalação

Para ter acesso ao arquivo de download, é preciso primeiro efetuar um cadastro junto ao site do fabricante. Feito isso, o download é liberado na própria página (ver seção **Links**). Um arquivo chamado ibcomparer.zip é disponibilizado.

Após realizar o download, descompacte o arquivo e execute IbComparerSetup.exe para iniciar a instalação da ferramenta. A instalação do DB Comparer é extremamente simples, de forma que não requer detalhes aqui.

No diretório de instalação, o arquivo IbComparer.exe corresponde à ferramenta gráfica do DB Comparer enquanto que o IbComparerC.exe diz respeito à aplicação console, por linha de comando. Além dos arquivos executáveis tem-se acesso ao arquivo de ajuda da ferramenta (IbComparer.chm).

Definindo os bancos de dados

Para realizar a comparação entre dois bancos de dados IB/FB, é preciso criar um projeto dentro da ferramenta. Isso pode ser feito acessando o menu File>New Project disponível na aplicação.

Nessa opção a tela de configuração do projeto é aberta e nela podemos definir os bancos de dados envolvidos na comparação.

Os termos utilizados são Master e Target, respectivamente mestre e destino.

Para demonstrar as funcionalidades e como utilizar o aplicativo, o banco de dados Employee.fdb foi copiado e em sua cópia a estrutura sofreu uma alteração. Dessa forma será possível comparar duas estruturas distintas. Ambos os bancos estão localizados dentro da pasta de instalação do Firebird. Quanto à versão do Firebird, para este exemplo é irrelevante. A **Figura 1** mostra a configuração realizada na janela Project Options para o novo projeto.

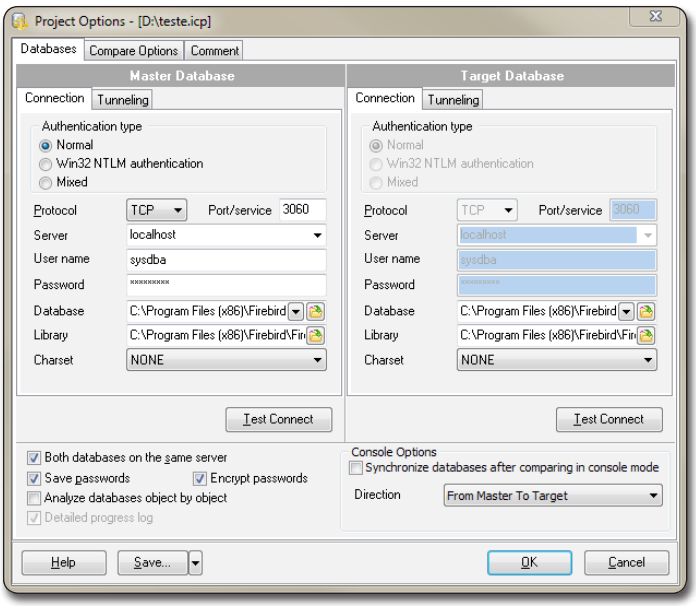


Figura 1. Configurando o acesso aos bancos de dados a serem comparados

Nota

Por padrão o campo Library vem definido como gds32.dll que é a biblioteca utilizada pelo InterBase. Para o Firebird altere o campo para a localização da fbClient.dll.

É possível testar a conexão com o banco de dados através do botão Test Connect de cada banco de dados. Ao marcar a opção Both databases on the same as configurações realizadas no servidor Master são replicadas para o Target.

O DB Comparer possui uma opção chamada Synchronize databases after comparing in console mode. Esta permite a sincronização automática entre os dois bancos de dados após a comparação dos mesmos.

Escolhendo o que comparar

É possível selecionar o que será comparado. Basta acessar a aba Compare Options e ao selecionar os tipos de objetos do banco (tabelas, views, etc) são exibimos mais detalhes de cada item. Assim é possível personalizar a comparação de acordo com as necessidades da situação (**Figura 2**).

A aba Comment é quase que auto explicativa. Nela podemos digitar um texto que explique o motivo da comparação de estruturas.

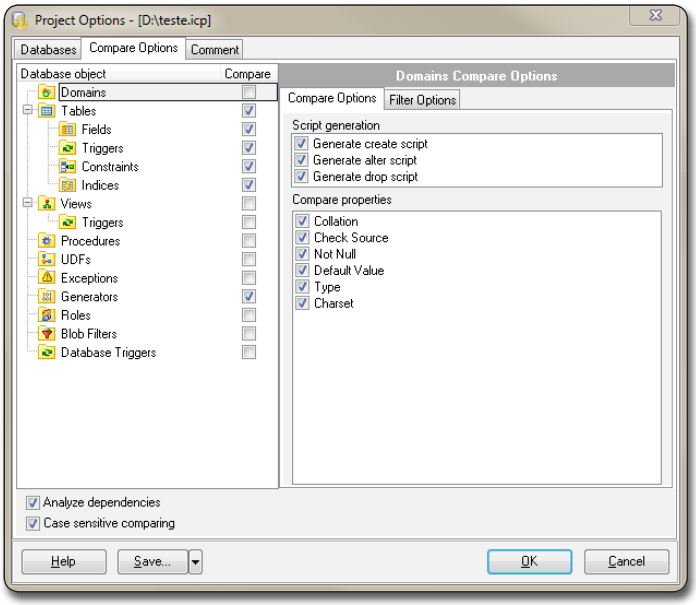


Figura 2. Definindo os objetos e propriedades a serem comparadas

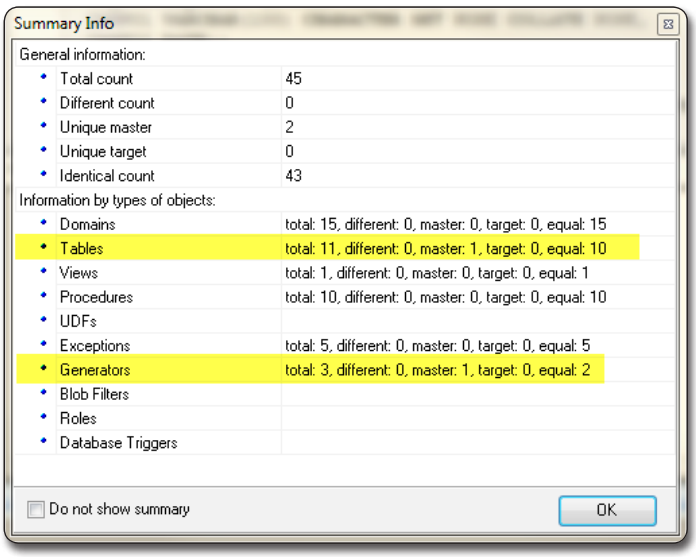


Figura 3. Sumário das diferenças

Uma vez tudo configurado, basta pressionar o botão OK para que o acesso aos bancos e comparação inicie. Ao terminar o processo é exibida uma janela de sumário (**Figura 3**), já indicando o que foi alterado.

Para otimizar a usabilidade, o projeto pode ser salvo através da opção File>Save Project. Será gerado um arquivo de extensão ICP que nada mais é do que um arquivo INI contendo as configurações definidas no projeto. Isso é muito útil, foi podemos configurar uma comparação uma única vez ou, ter configurações distintas para situações distintas.

Depois de realizada a comparação, é possível visualizar todos os objetos, de ambos os bancos, que precisam ser alterados/incluídos/

excluídos. Também é possível verificar os scripts necessários para realizar a sincronização entre o banco de dados Master e Target. A **Figura 4** mostra o DB Comparer após a comparação.

Customizando os scripts gerados

O resultado da comparação é exibido em um treeview que é agrupado principalmente pelas diferenças do master, pelas diferenças do target em relação ao master e ainda, mostra o que está idêntico entre os dois. A ferramenta permite navegar por esse treeview e quando um objeto é selecionado, a área da janela referente a exibição de scripts mostra o SQL necessário para ajustar o banco de dados.

Se for necessário incluir algo no script gerado, por exemplo, como um inserção de dados, é possível editar os scripts gerados durante a comparação, para isso basta realizar um duplo clique no script a ser alterado dentro da lista. A janela SQL Script Editor é aberta para editar o script (**Figura 5**).

Para a execução do script, é possível selecionar qual alteração executar ou, executar todas de uma única vez. A ferramenta disponibiliza o botão Execute All Scripts, que executará todos os comandos de alteração de estrutura.

Caso algum erro ocorra durante a execução o mesmo é mostrado ao usuário e o script é interrompido. Após a execução dos scripts o aplicativo exibe um diálogo que pergunta se deseja realizar outra comparação. Caso tenha perdido esse diálogo e deseje realizar uma atualização da comparação, basta pressionar a tecla F5. O processo de comparação será refeito, exibindo agora informações atualizadas.

Personalizações diversas

A opção Options>Keyboard Template permite a visualização de alguns templates disponíveis, que auxiliam no momento de digitação. Sua utilização é bastante simples, Basta digitar os caracteres de atalho e clicar na tecla espaço durante a digitação do script SQL. Por exemplo, para adicionar comentários ao script digite "///" e tecle espaço, aparecerá no editor os caracteres /* */ para que o comentário possa ser escrito.

No menu Options>Environment Options, é possível ajustar o comportamento da aplicação. Por exemplo, temos opções para criar uma configuração padrão de comparação, podemos deixar marcado apenas o item Tables. Assim, todo novo projeto já carregaria esse padrão. O editor SQL também pode ser ajustado para ter uma fonte de um determinado tipo, tamanho e inclusive as cores de destaque das palavras reservadas podem ser configuradas.

O visual geral do aplicativo pode ser configurado em Visual Options (**Figura 6**) que pode ser acessada através do menu Options>Visual Options. Nessa janela podemos selecionar diversos skins para serem aplicados ao IDE.

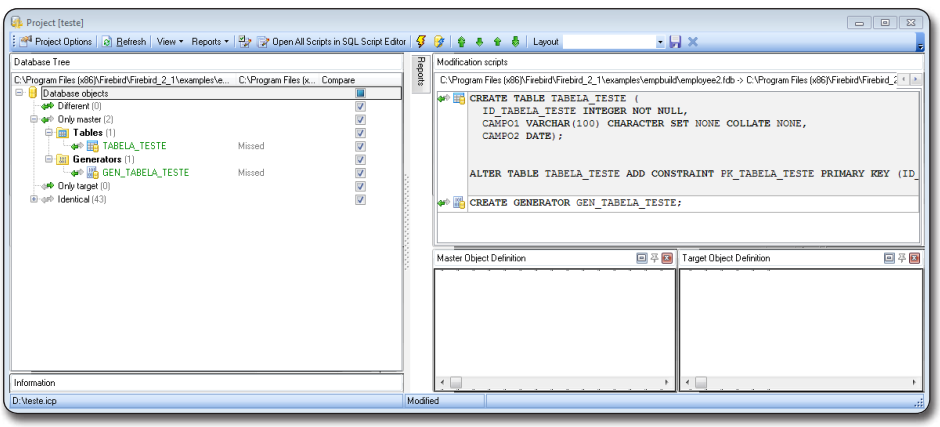


Figura 4. Diferenças encontradas nos bancos de dados Master e Target

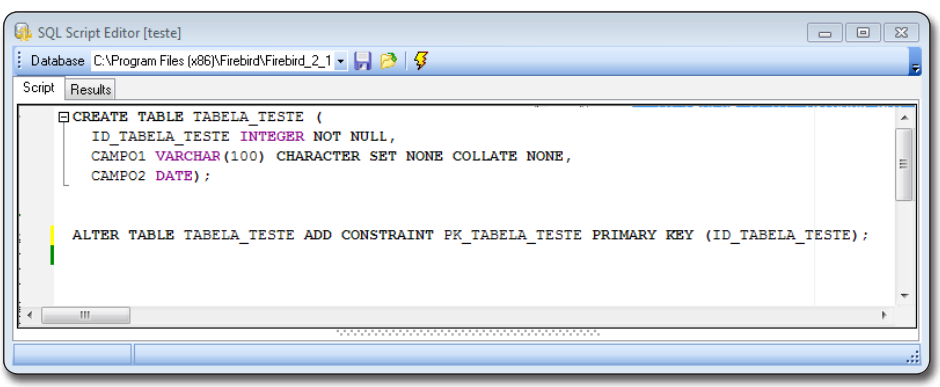


Figura 5. Alterando ou criando scripts DDL através do SQL Script Editor

Relatórios

Relatórios podem ser criados a partir das informações do projeto. Além disso, também é possível criar novos tipos de relatórios. Tudo isso está disponível na opção View>Reports. Por padrão a ferramenta traz alguns relatórios prontos, que sumarizam as alterações a serem realizadas, como mostra a **Figura 7**.

Para pode realizar a criação de relatórios personalizados, basta acessar o menu View>Reports e escolher a opção Create. O mais interessante desse recurso é que a ferramenta utiliza o componente Fast Report, que hoje é parte integrante do Delphi e conhecido da comunidade. Com o editor de criação de relatórios aberto, é possível identificar os dados que são disponibilizados. Temos informações para exibir os objetos que são diferentes entre os bancos de dados, detalhes das propriedades desses objetos e ainda informações que sumarizam o resultado da comparação (**Figura 8**). Como o editor não possuir restrição, é possível até criar um cabeçalho com o logo de sua empresa.

Durante o preview do relatório podemos salvá-lo através do botão Save Report ou

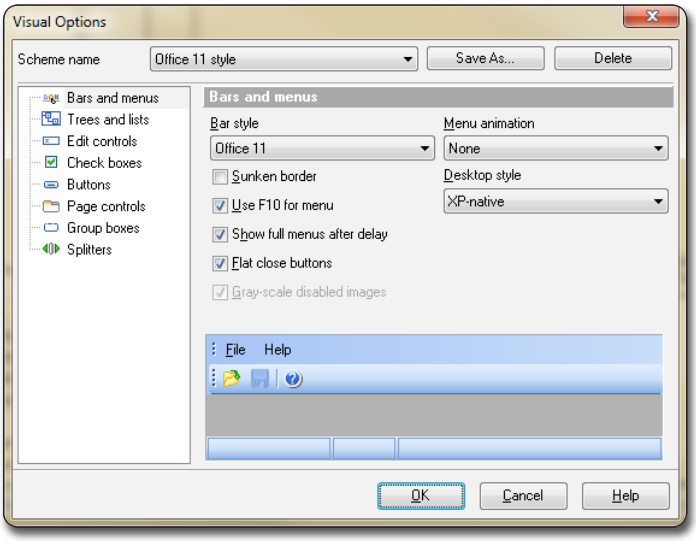


Figura 6. Visual Options

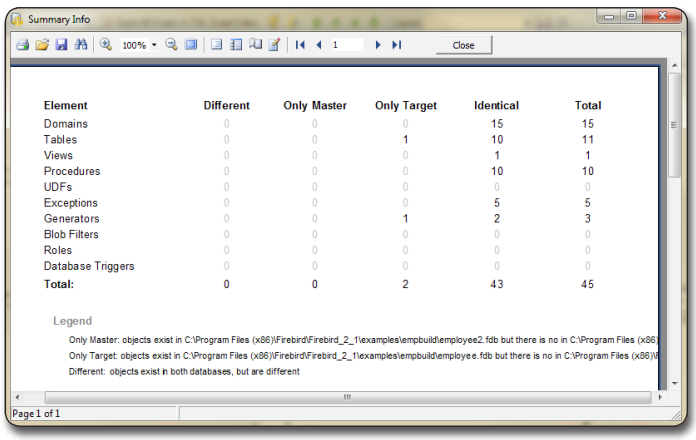


Figura 7. Relatório sumário

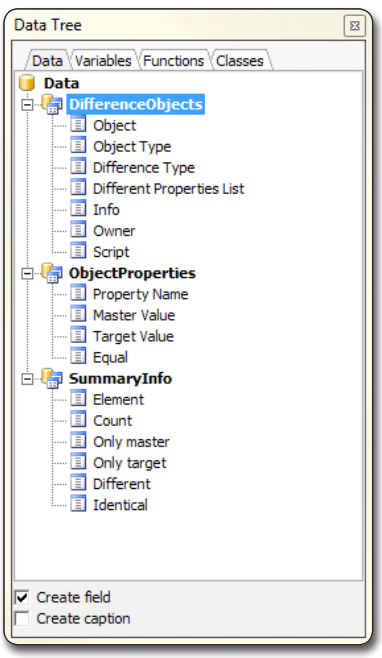


Figura 8. Informações disponíveis para criação do relatório

ainda imprimi-lo. Caso haja a necessidade de alterar a estrutura do relatório, é possível fazê-lo através da opção Edit disponível na barra de botões da janela Reports.

Como o DB Comparer utiliza o FastReport para edição e visualização de relatórios, os desenvolvedores que utilizam o FastReport em seus aplicativos Delphi não terão dificuldades em criar e alterar relatórios.

Aplicativo Console

Na pasta de instalação do DB Comparer temos o arquivo IbComparerc.exe. Este é o aplicativo console que pode ser utilizado para automatizar o processo de atualização de estruturas. Por exemplo, ele poderia ser agendado para ser executado a partir de um determinado horário.

A execução da aplicação é bastante simples. Ela recebe um primeiro parâmetro que informa qual é o arquivo de projeto que contém as configurações de comparação e quais bancos serão comparados. Um segundo parâmetro indica para qual arquivo serão gerados os scripts resultantes da comparação. Ou seja, a ferramenta irá comparar e precisa salvar essa diferença em algum lugar. Esse segundo parâmetro é para isso. Um terceiro parâmetro pode ser fornecido, mas é opcional. É o /E. Com ele indicamos á ferramenta que, logo após o término da comparação, o script gerado deve ser executado. Dessa forma, a seguir temos um exemplo de uso:

```
IbComparerc.exe C:\teste.icp scripts.sql /E
```

Comparação de dados

Mas, e se for necessário comparar além da estrutura, o conteúdo de uma tabela? O pessoal da EMS pensou nisso e desenvolveu também o aplicativo Data Comparer, que pode ser baixado do site da empresa (veja seção **Links**).

Após o devido download e instalação, que não requer detalhes, o processo todo é resumido em um assistente de oito passos.

Ao abrir o aplicativo temos o primeiro passo. A definição também do banco de dados de origem (source) e de destino (target). A tela é bem semelhante ao do aplicativo DB Comparer. Para realizar uma demonstração, vamos utilizar os bancos de dados citados anteriormente. Employee e sua cópia Employee2. Contudo, no Employee2 foi incluído um novo registro na tabela Country, foram incluídas informações sobre o Brasil.

Mapeando tabelas

Não é necessário ter o mesmo banco de dados como source e target, é possível realizar uma comparação com uma base de dados diferentes, visto que é preciso determinar as correspondências de tabelas e campos, como mostra a **Figura 9**.

Ao selecionar na coluna Target Tables, na linha Country, a tabela Country de destino, automaticamente na parte inferior da janela, os campos são identificados. Um recurso que merece atenção é que se pode indicar um filtro de dados, para restringir a comparação. Por exemplo, se fossemos comparar uma tabela de

clientes, poderíamos limitar a comparação apenas entre os clientes do tipo "pessoa jurídica". Às vezes isso é importante porque a comparação de dados varre os registros das tabelas, procurando por diferenças, o que exige tempo.

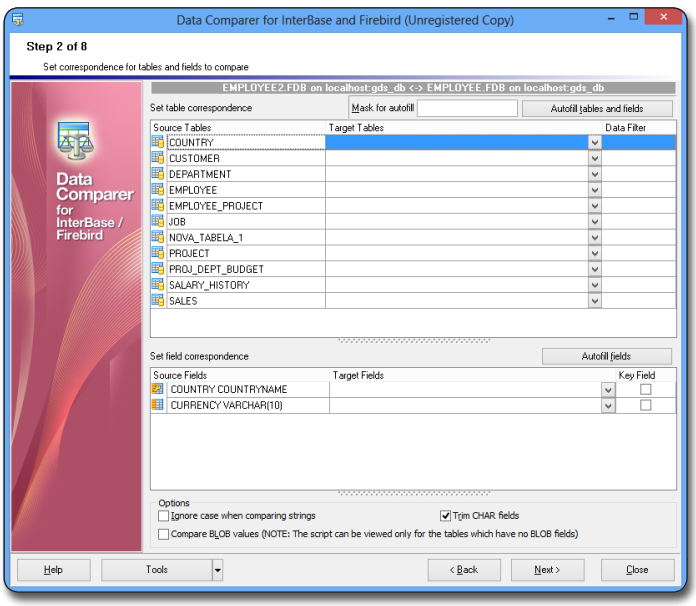


Figura 9. Mapeamento de tabelas e campos

Esse foi o segundo passo. Em Next chegamos ao terceiro e a comparação é realizada. Ao término temos um sumário por tabela (**Figura 10**), mostrando o que está idêntico, o que está diferente, faltante e o que precisa ser adicionado. Lembrando que isso sempre na direção da origem para destino.

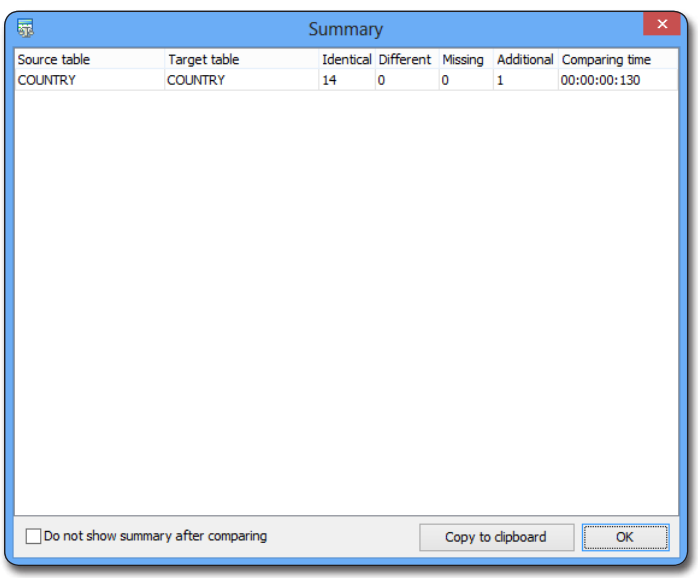


Figura 10. Resumo da comparação

Ao fechar a janela de sumário, vemos origem e destino lado a lado (**Figura 11**). Então é possível selecionar qual das diferenças será aplicada ou, exportar os dados da tabela. No botão Export é possível configurar quais registros serão exportados, os faltantes, os diferentes, etc. e para qual tipo de arquivo irão: Excel, RTF ou HTML. Isso é importante, pois em caso de dúvida é melhor exportar para um arquivo temporário e verificar as informações.

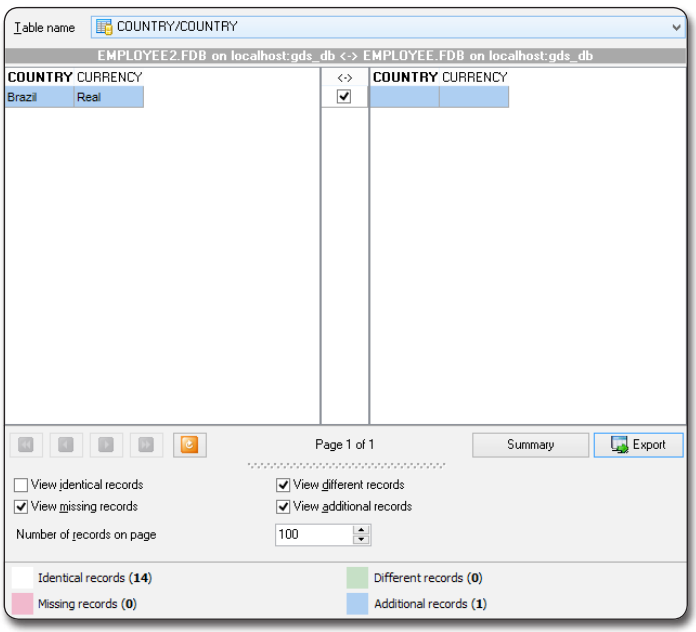


Figura 11. Visualização do que está diferente

Sincronização

Ao avançar, passo 4, configuramos a sincronição dos dados. É possível sincronizar para uma tabela nova no destino. Bastando escolher a opção Synchronize in new table e informar o nome dessa nova tabela. Além disso também pode limitar o que será sincronizado, como inserir registros adicionais, excluir registros faltantes, atualizar registros diferentes e até mesmo, excluir e recriar índices.

Como o volume de dados pode ser grande, existe a configuração de transações. Nela indicamos um lote de registros para realização do commit. Ou seja, de quanto em quanto será commit. O valor padrão é de a cada 100 registros.

Com as configurações padrão, o assistente do aplicativo ao avançar para no passo 6, que exibe o script SQL de alteração.

Ao continuar temos uma funcionalidade muito importante. É possível carregar outro script SQL que será executado antes da sincronização e outro para depois. Assim, é possível realizar qualquer preparação necessária de forma automática. Avançando temos a janela de log, com o botão Synchronization. Ao clicar nele o processo é concluído (**Figura 12**).

Conclusão

É muito importante conhecer ferramentas que podem nos auxiliar no dia a dia com tarefas que não são o foco de nosso trabalho. Por exemplo, existem pessoas que preferem criar todo um banco de dados escrevendo sentença SQL por sentença SQL, não utilizam

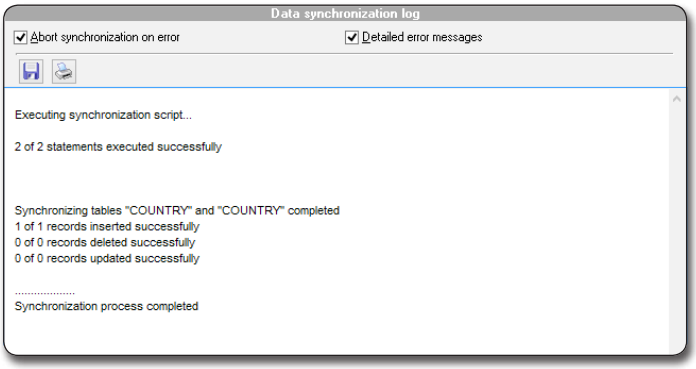


Figura 12. Log de sincronização

nenhum designer para tornar o processo mais rápido. O mesmo serve para as operações mostradas nesse artigo. Porque criar um utilitário a parte, você mesmo, para compara sua base de dados e seu conteúdo se já temos isso pronto nas ferramentas da EMS? Claro, elas não são as únicas. O ponto aqui é, otimização de tempo. Fica aqui a dica.

Paulo Quicoli
pauloquicoli@gmail.com
Editor da revista ClubeDelphi e editor técnico .Net Magazine. Formado em processamento de dados pela FATEC-TQ. Atua como analista programador de sistemas da Siplan Control-M (www.siplancontrolm.com.br) unidade Jaboticabal, e professor da FATEC, campus Taquaritinga.
Blog: http://quicoli.worpress.com

http://www.sqlmanager.com.br/html/db_comparer_para_interbase.html

http://www.sqlmanager.net/en/products/ibfb/datacomparer

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Gerenciamento automático de metadados

Um framework automatizado para atualizar estruturas de BDs

Em desenvolvimento de software, o crescimento de uma empresa se dá no aumento de usuários de seus sistemas. Muitas vezes acontece algum evento que estimula um crescimento acima do esperado, muitas vezes fazendo com que sua carteira de clientes dobre em questão de meses. Isso afeta toda a empresa, pois mais trabalho é requerido, de todos os departamentos. É nesse momento que a empresa deve se manter sólida e manter sua qualidade.

O termo qualidade na área de software é bem abrangente, envolve desde entrega no prazo, custos dentro do orçamento e claro, a satisfação do cliente. Com um crescimento repentino, ações que muitas vezes eram tomadas para cinco ou dez clientes, não são mais praticáveis, pois agora pode se ter cinquenta ou cem clientes.

Um exemplo é a atualização de versão de sistema no que tange a atualização de estrutura de banco de dados. Quando se tem poucos clientes, é muito fácil realizar um acesso remoto, realizar uma comparação de banco de dados (produção vs. desenvolvimento) e atualizar manualmente essa estrutura (utilizando softwares de comparação de base de dados). Mas, quando o volume de clientes cresce, o tempo gasto nessas operações é precioso, se antes, com dez clientes despendíamos 10 minutos em cada atualização, teríamos um montante de 1h e 40min em um processo geral de atualização. Mas, com 100 clientes temos outra realidade: mais de 16h nesse mesmo processo. Mas, qual a relação disso com qualidade? A satisfação de seus clientes. Antes, eles estavam acostumados a no máximo, esperar cerca de 1h, agora, quase dois dias. É preciso criar então ações para garantir a qualidade de seu software.

Automação de atualização de estrutura

Vamos apresentar neste artigo um pequeno framework que irá gerenciar as alterações de estrutura de uma base de dados e irá atualizar de forma automática essa estrutura. Esses são os requisitos básicos, mas temos outros importantes como: funcionar tanto para IBX quanto para dbExpress, poder ser utilizado por projetos a partir do Delphi 7, deve manter um histórico de toda alteração que o banco de dados sofreu e deve permitir a execução de blocos SQL sob um controle transacional, ou seja, se algum comando SQL não puder ser executado, toda a atualização de estrutura é desfeita. Esse acontecimento deve ser notificado. Outro requisito importante é sobre sua reutilização. Ele não terá dependência com projeto algum e sua integração com projetos existentes será bem simples.

Resumo DevMan

Cenário:

Neste artigo abordamos a automatização de um processo que pode custar tempo precioso durante a atualização de um sistema em um cliente. A atualização da estrutura de banco de dados. Esse processo começa a "incomodar" quando se tem muitos clientes, fazendo com que a atualização de uma versão leve mais tempo do que o esperado.

Esse cenário é principalmente visto onde se tem uma equipe de suporte, sem muito conhecimento técnico, e é preciso realizar uma atualização de versão do seu sistema que implica na atualização da estrutura da base de dados do cliente. Isso pode causar certa dificuldade e demora para a equipe de suporte, impactando no tempo utilizado para realizar a mudança de versão. A solução é simples, um pequeno framework que mantém os scripts a serem aplicados, verifica qual versão necessita ser instalada, a instala e efetua um feedback através de eventos.

Em que situação o tema é útil:

Quando se deseja atualizar de forma automática a estrutura de uma base de dados, sem intervenção humana, para ganho de tempo com o processo.

Tutorial

Projeto teste

Para conter o projeto foi criada uma pasta AtualizadorBanco, nela foi criado um banco de dados Firebird 1.5 de nome TESTE. Então no Delphi criamos um novo projeto VCL e o salvamos como ProjetoTeste. Não mudamos o nome do formulário principal porque isso não influencia nos testes e nele, inserimos controles que podem ser vistos na **Figura 1**. O único controle que teve seu nome alterado foi o TMemo, para memoEventos.

Nota

Para controles visuais que são acessados via código, é uma boa prática alterar sua propriedade Name para algo significativo, relacionado ao contexto de seu uso.

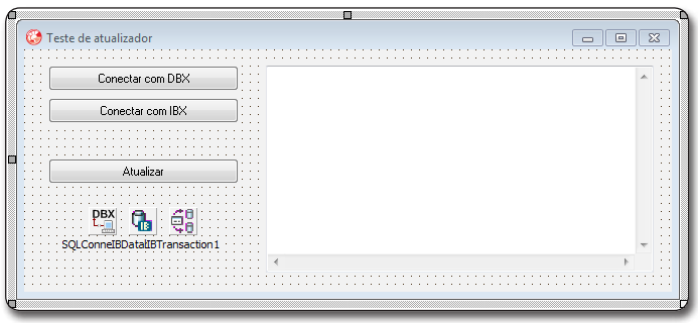


Figura 1. Aplicativo de teste

A ideia geral de funcionamento é conectar no banco de testes, seja via dbExpress ou via IBX e ao clicar em Atualizar, observar a resposta do que é feito no TMemo.

Classes de apoio

O que justifica uma atualização de aplicação em um cliente é a disponibilidade de uma nova versão. Assim, esse conceito foi abstraído em uma classe, chamada TVersao. Adicionamos ao projeto uma unit de nome devmedia.Atualizador.pas para conter essa classe que pode ser vista na **Listagem 1**.

TVersao representa uma versão do banco de dados, contendo um Id numérico que indica a ordem sequencial da evolução de estrutura, uma propriedade Numero, do tipo string, utilizada para relacionar a alteração de estrutura com uma versão de sistema. Isso pode ser utilizado para responder questões como, quais versões do sistema causaram uma alteração no banco de dados? Temos também uma propriedade Data do tipo TDateTime, para conter a data dessa versão, Detalhes, do tipo string para conter uma descrição do que será alterado. Nessa classe também estará contido o script SQL que aplica as mudanças no banco de dados, na propriedade Script do tipo TStringList. Por fim, temos a propriedade ExigeApenasUmUsuario, um boolean que indica a necessidade de se ter apenas um usuário conectado para se aplicar a atualização de versão. O construtor e destrutor respectivamente apenas instanciam e liberam da memória o objeto TStringList utilizado pela propriedade Script.

Listagem 1. Classe TVersao

```
TVersao = class(TPersistent)
private
  ..
public
  constructor Create;
  destructor Destroy;
  property Id: Integer read FId write SetId;
  property Numero: string read FNumero write SetNumero;
  property Data: TDateTime read FData write SetData;
  property Script: TStringList read FScript write SetScript;
  property Detalhes: string read FDetalhes write SetDetalhes;
  property ExigeApenasUmUsuario: boolean read FExigeApenasUmUsuario
  write SetExigeApenasUmUsuario default false;
end;
```

Ao longo do ciclo de vida de um projeto, várias modificações podem ser feitas em um banco de dados, isso vem de encontro com um dos requisitos, manter o histórico dessas modificações. Como cada modificação está representada pelo tipo TVersao, é preciso termos uma lista que mantenha objetos desse tipo. Atualmente a forma mais fácil de alcançar isso, seria ter um objeto do tipo TList<TVersao>, fazendo uso dos recursos mais atuais, no caso, Generics. Contudo esbarramos em outro requisito, o framework também deve ser compatível com o Delphi 7. Então é necessário empregar uma solução pré Generics. Criar uma classe que herde do tipo TObjectList. Esse tipo é uma lista de objetos, mais especificamente de objetos do tipo TObject. Como TObject é o tipo base de todo objeto em Delphi, qualquer objeto poderia ser adicionado a uma lista desse tipo, o que pode levar a eventuais erros no desenvolvimento. Mas, ao herdar a partir dela, podemos "tipá-la", fazendo com que somente aceite e retorne objetos do tipo TVersao. Na **Listagem 2** temos a interface da classe TListaVersao.

Listagem 2. Classe TListaVersao

```
TListaVersao = class(TObjectList)
private
  function GetObject(Index: Integer): TVersao;
  procedure SetObject(Index: Integer; const Value: TVersao);
protected
  procedure Insert(Index: Integer; Obj: TVersao);
public
  function Add(Obj: TVersao): Integer;
  property Objects[Index: Integer]: TVersao read GetObject
  write SetObject; default;
end;
```

A implementação dos métodos é bastante simples, simplesmente executamos os métodos da classe base, mas nos métodos que retornam objetos, por exemplo, GetObject, realizamos um typecast. Isso porque a lista continua sendo de TObject, assim, todo objeto armazenado, é convertido para seu tipo base mais ancestral, o TObject, e no momento de recuperar um objeto, o que retorna da lista é um TObject, por isso a necessidade desse typecast, como mostra o código a seguir.

```
function TListaVersao.GetObject(Index: Integer): TVersao;
begin
  Result := inherited GetItem(Index) as TVersao ;
end;
```

Existem duas formas de realização de typecast. A primeira, realizada pelo operador **as** pode ser avaliada em tempo de execução. Isso quer dizer que, se tenho uma classe TPessoa e tentar convertê-la para TProduto (que não faz parte de sua hierarquia), em tempo de execução será levantada uma exceção dizendo que um TPessoa não pode ser um TProduto. A segunda forma é a conversão direta utilizando a seguinte sintaxe: TClasse1(Objeto). Transformamos, sem verificação alguma, a instância Objeto em um objeto do tipo TClasse1. Se isso não for possível, erros de memória acontecerão, prejudicando o funcionamento do sistema.

Outra exigência é a de notificar caso um erro aconteça. Mas essa notificação não pode estar vinculada a projeto algum, ou seja, não posso simplesmente exibir uma caixa de mensagem a partir do framework. Assim, a solução é a criação de alguns eventos personalizados.

Eventos

Um evento é um mecanismo que liga um acontecimento a um código. Sendo mais técnico, um evento é um ponteiro de método que aponta para um método específico de uma instância de classe. Do ponto de vista de um desenvolvedor iniciante, um evento é apenas um nome relacionado a um acontecimento no sistema, como OnClick, que pode receber código que trata dessa ocorrência. Quando adicionamos um TButton em um formulário qualquer e efetuamos um clique duplo sobre ele, o IDE do Delphi automaticamente implementa no formulário o manipulador (handler) do evento OnClick. Assim, quando ocorre o evento click nesse botão, o método associado no OnClick é executado. Dessa forma temos a sequência exibida na **Tabela 1**.

Ocorrência ->	Evento ->	Manipulador
clique no Button1	Button1.OnClick aponta para Form1.Button1Click	Form1.Button1Click é executado

Tabela 1. Sequência de um evento

Podemos afirmar então que,

- Eventos são ponteiros para métodos;
- Eventos são propriedades;
- Tipos manipuladores de evento são procedures;
- Manipuladores são opcionais.

Eventos são ponteiros para métodos

Um ponteiro de método é um tipo especial que aponta para um método específico de um objeto (instância de classe). Quando um manipulador é associado a algum evento de algum componente, essa associação não é apenas pelo nome do método, mas é feita para uma instância específica do objeto manipulador. No exemplo anterior, do click do botão que consta em um formulário, esse objeto manipulador seria o formulário. Mas esse objeto não precisa ser somente do tipo TForm.

Eventos são propriedades

Componentes usam propriedades para expor eventos. Diferentemente de uma propriedade comum, um evento não utiliza métodos Get/Set para implementar sua leitura e acesso. Eles acessam de forma direta um campo privado do mesmo tipo da propriedade. Por convenção, o nome desse campo é o nome da própria propriedade, precedido pela letra F. Por exemplo, o ponteiro de método do evento OnClick é armazenado em uma campo chamado FOnClick, do tipo TNotifyEvent (um tipo padrão de evento) e possui a seguinte declaração:

```
type
 TControl = class(TComponent)
 private
  { campo para armazenar o ponteiro de método }
  FOnClick: TNotifyEvent;
  ...
 protected
  property OnClick: TNotifyEvent read FOnClick write FOnClick;
 end;
```

Como qualquer outra propriedade, é possível setar e mudar o valor de um evento em tempo de execução.

Tipos manipuladores de evento são procedures

Para compreender esse assunto, é bom analisar o exemplo de passagem de parâmetros para um manipulador do evento OnKeyPress, que é do tipo TKeyPressEvent. Esse tipo define dois parâmetros, um para indicar qual objeto gerou o evento e outro para indicar qual tecla foi pressionada:

```
type
 TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;
```

Normalmente, o parâmetro Key contém o caractere pressionado pelo usuário. Mas sob certas circunstâncias, contudo, o usuário do componente quer mudar esse caractere. Por exemplo, transformando tudo o que é digitado para maiúsculo. Assim, como o manipulador é uma procedure, do tipo especificado pelo evento, ele deve ter a mesma assinatura para receber os parâmetros que o evento irá disparar:

```
procedure TForm1.Edit1KeyPressed(Sender: TObject; var Key: Char);
begin
  Key := UpCase(Key);
end;
```

Manipuladores são opcionais

Antes de se disparar um evento, é preciso verificar se seu manipulador foi associado, pois não existe obrigatoriedade de se responder a todos os eventos. Por exemplo, temos o evento OnMouseMove, ele é disparado sempre que o mouse se move, contudo, apenas quando um manipulador está associado a ele.

Eventos personalizados

Uma vez que compreendemos o que são eventos, no framework foram definidos três:

```
TErroExecucaoScript = procedure(Sender: TObject; Versao: TVersao;
Excecao: Exception) of object;
TAplicacaoVersao = procedure (Sender: TObject; Versao: TVersao) of object;
TMaisDeUmUsuario = TNotifyEvent;
```

TErroExecucaoScript é disparado quando acontece uma falha na execução de um script de atualização. Assim, pelos parâmetros (Versao e Excecao) passados pode-se mostrar ao usuário a versão que está com problema e o erro ocorrido. TAplicacaoVersao é disparado quando uma versão é concluída, quando foi instalada com sucesso. Como o parâmetro Versao é passado, é possível registrar a versão instalada. Por fim, TMaisDeUmUsuario é disparado apenas quando a versão a ser instala requer apenas um usuário conectado, permitindo ao desenvolvedor tratar a situação. Todos esses eventos são utilizados em uma classe chamada TAtualizadorBd.

O núcleo

A classe TAtualizadorBd pode ser chamada de núcleo do framework, porque ela é a que coloca outras partes para funcionar, integrando todo o processo de atualização. A **Listagem 3** mostra seu código parcial e através dele serão apresentadas outras partes do framework.

Listagem 3. Classe TAtualizadorBd

```
TAtualizadorBd = class(TPersistent)
private
  ...
public
  constructor Create(ConectorBanco: TConector; CharIndicadorBlocoComando:
   string = ';');
  destructor Destroy; override;
  procedure RegistrarVersao(const versao: TVersao);
  function ObterListaScripts: TListaVersao;
  procedure AplicarAtualizacoes(versoes: TListaVersao);
  property OnErroExecucaoScript: TErroExecucaoScript read FOnErroExecucao
  Script write SetOnErroExecucaoScript;
  property OnAplicacaoVersao: TAplicacaoVersao read FOnAplicacaoVersao
  write SetOnAplicacaoVersao;
  property OnMaisDeUmUsuario: TMaisDeUmUsuario read FOnMaisDeUm
  Usuario write SetOnMaisDeUmUsuario;
end;
```

São expostos os métodos que o desenvolvedor irá utilizar em sua aplicação para realizar a atualização de banco de dados.

Ao instanciar um TAtualizadorBd é preciso passar em seu construtor um objeto do tipo TConector. Nesse momento vamos dizer que ele é o responsável por abstrair a conexão com o banco de dados, de tal forma que o framework fique independente de camada de acesso. Posteriormente teremos mais detalhes dessa classe. Também indicamos no construtor o caractere que indica blocos de comando SQL. Vamos entender essa necessidade.

Um script SQL pode conter várias instruções de atualização, como criação de tabelas, criação de índices, alteração de campos e tudo mais. Essas instruções são tratadas como blocos de comando para o framework e cada bloco é executado individualmente, contudo, toda a execução da versão ocorre dentro de um contexto transacional.

O método RegistrarVersao recebe como parâmetro um objeto TVersao. Esse objeto é então armazenado em uma lista para posterior identificação do que deve ser executado, qual versão deve ser aplicada. Essa identificação é feita no método Obter-ListaScripts, que retorna um objeto do tipo TListaVersao, definido anteriormente. A **Listagem 4** mostra sua implementação.

Listagem 4. Método RegistrarVersao

```
01 function TAtualizadorBd.ObterListaScripts: TListaVersao;
02 var
03   versaoInstalada: TVersao;
04   listaFiltrada: TListaVersao;
05   i: Integer;
06 begin
07   try
08     versaoInstalada := FConector.RetornaUltimaVersaoInstalada;
09   except
10     listaFiltrada := TListaVersao.Create(True);
11     Result := listaFiltrada;
12     Exit;
13   end;
14
15   if versaoInstalada = nil then
16     listaFiltrada := FListaVersoes
17   else
18   begin
19     listaFiltrada := TListaVersao.Create(True);
20     for i:= 0 to FListaVersoes.Count-1 do
21     begin
22       if FListaVersoes[i].Id > versaoInstalada.Id then
23         listaFiltrada.Add(FListaVersoes[i]);
24     end;
25   end;
26   result := listaFiltrada;
27 end;
```

A comparação se dá pela identificação da última versão instalada e listagem das versões que foram instaladas. A classe TConector disponibiliza o método RetornaUltimaVersaoInstalada que retorna o último objeto TVersao instalado. Esse objeto fica armazenado na variável versaoInstalada (linha 8). Caso um erro aconteça nesse momento, criamos uma lista vazia e a retornamos. Isso posteriormente é utilizado para que em caso de erro, nada seja executado. Contudo, não aconteceu erro algum, verificamos se alguma versão já foi instalada (linha 15). Se nada foi instalado ainda, então o método deve retornar a lista que contém todas as versões registradas. Esse é o cenário da primeira execução, quando nada ainda foi registrado como executado. Mas, quando alguma versão já foi aplicada, esse

objeto não é **nil**, assim, filtramos a lista das versões registradas, deixando as que são posteriores a ultima instalada (linhas 20 a 24). Essa identificação se dá pelo Id da propriedade TVersao. Por isso é importante que se mantenha uma sequência única e progressiva desse valor. Finalizada a filtragem, essa lista é retornada para posterior processamento. Esse processamento é feito pelo método AplicarAtualizacoes, que como sugere o nome, aplica as versões de atualização. Isso pode ser visto na **Listagem 5**.

Listagem 5. Método AplicarAtualizacoes

```
01 procedure TAtualizadorBd.AplicarAtualizacoes(versoes: TListaVersao);
02 var
03   i: Integer;
04 begin
05   for i:=0 to versoes.Count-1 do
06   begin
07     try
08       if versoes[i].ExigeApenasUmUsuario then
09       begin
10         if FConector.MaisDeUmConectado then
11         begin
12           if Assigned(FOnMaisDeUmUsuario) then
13             OnMaisDeUmUsuario(Self);
14           Break;
15         end;
16       end;
17
18       FConector.ExecutarScript(versoes[i]);
19       if (Assigned(FOnAplicacaoVersao)) then
20         OnAplicacaoVersao(Self,versoes[i]);
21     except on e: Exception do
22       begin
23         if Assigned(FOnErroExecucaoScript) then
24           OnErroExecucaoScript(self,versoes[i], e);
25         Break;
26       end;
27     end;
28   end;
29 end;
```

O método recebe como parâmetro a lista de versões que deve ser aplicada, ou seja, somente as versões posteriores à última não instalada. Essa lista é percorrida (linha 5) e para cada elemento, que é um objeto TVersao, é verificado se ele exige que apenas um usuário esteja conectado (linha 8). Exigindo, o evento OnMaisDeUmUsuario deve ser disparado, para que o desenvolvedor trate isso em sua aplicação. Contudo, o evento só é disparado caso tenha sido registrado um manipulador para o evento. Isso é verificado pela função **Assigned** (linha 23). Mas, independente do evento possuir um manipulador, o processo é interrompido (linha 14). Quando essa identificação de usuários conectados não é necessária, chamamos o objeto conector, que foi passado via construtor, para executar seu método ExecutarScript, passando como parâmetro o objeto TVersao corrente. Se o evento OnAplicacaoVersao possuir um manipulador, o mesmo é disparado (linha 20) para que o desenvolvedor possa responder adequadamente. Se algum erro acontece nessa execução o evento OnErroExecucaoScript pode ser disparado (linha 24) para que o desenvolvedor exiba isso em seu sistema.

Conexão com um banco de dados

O framework aplica atualizações em um banco de dados, para tal, precisa se conectar a esse banco. Para não deixar o framework rígido, isto é, preso a um único banco de dados ou a uma única tecnologia de acesso, essa conexão foi abstraída em uma classe. Isso obedece a uma boa prática da orientação a objetos: encapsular o que varia. A classe TConector (**Listagem 6**) encapsula essa variação de banco de dados e de tecnologia de acesso.

Listagem 6. Classe TConector

```
TConector = class
private
  ..
public
  property CharBlocoComando: string read FCharBlocoComando write
  SetCharBlocoComando;
  procedure VerificaCriaEstruturaBasica; virtual; abstract;
  function MaisDeUmConectado: Boolean; virtual; abstract;
  function RetornaUltimaVersaoInstalada: TVersao; virtual; abstract;
  procedure ExecutarScript(versao: TVersao); virtual; abstract;
end;
```

Ela funciona como um contrato, determinando métodos virtuais e abstratos que serão implementados por classes específicas. Essas classes herdam de TConector e implementam cada métodos utilizando uma tecnologia específica. Contudo, a classe TAtualizadoBd conhece apenas TConector, que é passado em seu construtor. Assim, podemos passar qualquer descendente de TConector e o framework todo irá funcionar. Isso é possível graças à característica de polimorfismo e herança que a orientação a objetos fornece. Em um ponto onde se espera uma classe pai, podemos passar sua filha e a mesma será aceita.

Voltando aos requisitos do framework, é necessário que o mesmo funcione com dbExpress, assim, foi adicionada uma nova unit ao projeto, chamada de devmedia.ConectorDBXXE2. Essa unit contém a implementação específica de TConector que utiliza a versão mais recente do dbExpress. A **Listagem 7** mostra a declaração da classe TConectorDBXXE2.

Listagem 7. Classe TConectorDBXXE2

```
TConectorDBXXE2 = class(TConector)
private
  FConexao: TSQLConnection;
public
  constructor Create(const objConexao: TSQLConnection); reintroduce;
  procedure VerificaCriaEstruturaBasica; override;
  function MaisDeUmConectado: Boolean; override;
  function RetornaUltimaVersaoInstalada: TVersao; override;
  procedure ExecutarScript(versao: TVersao); override;
end;
```

É possível identificar a herança realizada a partir de TConector. Outro detalhe fica por conta dos métodos herdados, que estão sinalizados com modificador **override**, indicando que estão sobrepondo qualquer funcionalidade existente da classe pai. Porém temos um método novo, na verdade, um construtor.

Nota

Na seção uses são adicionadas as seguintes referências necessárias: SysUtils, Classes, Contnrs, Forms, devmedia.AtualizadorBanco, SqlExpr, Data.DBXInterBase, Data.FMTBcd, Data.DB. A unit devmedia.AtualizadorBanco é referenciada para poder acessar os tipos e eventos nela definidos.

Esse construtor, Create, recebe como parâmetro um objeto do tipo TSQLConnection, que é o componente de conexão do dbExpress. Esse objeto de conexão será utilizado posteriormente para executar os scripts. Observe também que o construtor foi marcado com o modificador **reintroduce**. Isso oculta o construtor da classe pai, fazendo com que apenas esse seja disponível.

Tabela obrigatória

O framework registra quais versões foram instaladas. Para isso ele cria no banco de dados uma tabela específica, através do método VerificaCriaEstruturaBasica, como mostra a **Listagem 8**.

Esse método é chamado a partir do método RetornaUltimaVersaoInstalada, para garantir que o banco de dados tenha a estrutura mínima. Entre as linhas 9 e 13 instanciamos um objeto TSQLDataSet, que permite a execução de sentenças SQL. Esse objeto então é configurado para utilizar a conexão que foi anteriormente passada via construtor. A sentença a ser executada realizada uma pesquisa nas tabelas de sistema do Firebird, procurando por uma tabela chamada "VERSIONAMENTO". Se essa tabela não existir, seu retorno é zero e então é necessário criá-la. Essa criação acontece dentre de uma transação, representada pelo objeto Transacao. Nele são especificadas as características dessa transação, que são um Id e o nível de isolamento. O Id é apenas um identificar, que não pode repetir-se, já o nível, é um valor que expressa o tipo de transação a utilizar. No caso utilizamos o Read Committed, que indica que apenas transações já comitadas serão lidas. Então na linha 29 iniciamos a transação e na 30, através do método ExecuteDirect do objeto de conexão, executamos a sentença SQL que cria o tabela necessária. Logo após um commit é efetuado, para que a alteração seja persistida no banco e na sequência, outro SQL é executado para criação da chave primária da tabela. Caso um erro aconteça nesses procedimentos, um rollback é efetuado (linha 41), desfazendo qualquer alteração que essa transação tenha iniciado.

Obtendo a última versão instalada

O método RetornaUltimaVersaoInstalada possui essa responsabilidade, retornando um objeto TVersao. A **Listagem 9** mostra seu código.

A linha 14 contém o SQL que recupera o registro da última versão instalada. De posse dessa informação, um objeto TVersao é instanciado e como o que importa é apenas o Id dessa versão o objeto não é preenchido em sua totalidade, apenas o Id, Numero e Data são retornados (linhas 18 a 21). Isso é o suficiente.

Execução do script

Isso fica por conta do método ExecutarScript que recebe como parâmetro o objeto TVersao a ser executado. A **Listagem 10** exibe seu conteúdo.

Como podemos receber um script longo, é preciso quebrar seus blocos de comandos. Isso é feito entre as linhas 11 e 23. O script é percorrido linha a linha. Se a linha atual possui o caractere que indica bloco de comandos, esse caractere é excluído e a linha é adicionada em um objeto TStringList. Se a linha não possui o caractere indicador, seu conteúdo é concatenado em uma variável string, para formar uma sentença até que se encontre o próximo

Listagem 8. Método VerificaCriaEstruturaBasica

```
01 procedure TConectorDBXXE2.VerificaCriaEstruturaBasica;
02 var
03   qry: TSQLDataSet;
04   Transacao: TTransactionDesc;
05 begin
06   if not FConexao.Connected then
07     FConexao.Open;
08
09   qry := TSQLDataSet.Create(nil);
10   qry.SQLConnection := FConexao;
11   qry.CommandType := ctQuery;
12   qry.CommandText :='select 1 as existe from rdb$relations
       where rdb$relation_name = ' + QuotedStr('VERSIONAMENTO');
13   qry.Open;
14
15   try
16     if qry.FieldByName('existe').AsInteger = 0 then
17     begin
18       if MaisDeUmConectado then
19       begin
20         qry.Close;
21         FreeAndNil(qry);
22         raise Exception.Create('Mais de um conectado');
23         Exit;
24       end;
25
26     try
27       Transacao.TransactionID := 1;
28       Transacao.IsolationLevel := xilREADCOMMITTED;
29       FConexao.StartTransaction(Transacao);
30       FConexao.ExecuteDirect('CREATE TABLE VERSIONAMENTO ( ' +
31                   'ID_VERSIONAMENTO   INTEGER NOT NULL,' +
32                   'NRO                VARCHAR(15),' +
33                   'DATA               DATE,' +
34                   'APLICADA           INTEGER' +
35                   ')');
36       FConexao.Commit(Transacao);
37       FConexao.StartTransaction(Transacao);
38       FConexao.ExecuteDirect('ALTER TABLE VERSIONAMENTO ADD
         CONSTRAINT PK_VERSIONAMENTO PRIMARY KEY (ID_VERSIONAMENTO)');
39       FConexao.Commit(Transacao);
40     except
41       FConexao.Rollback(Transacao);
42     end;
43   end;
44 except
45 end;
46 end;
```

caractere indicador e seja adicionada no TStringList. Na sequência, uma vez terminado esse processo de separação, o objeto TStringList, no código representado pelo objeto comando, é percorrido e cada linha, que agora é uma instrução completa, é executada pelo método ExecuteDirect do objeto de conexão (linhas 28 a 38). Antes que esse processo ocorra, uma transação é iniciada (linha 27), assim, como a execução da sentença ocorre dentro de um bloco try..except, caso aconteça a exceção efetuamos um rollback na transação e disparamos uma exceção. Mas, se nenhum erro acontece, a cada passo do comando **for** é feita uma verificação para identificar se estamos na última sentença a ser executada. Estando, comitamos a execução das sentenças (linhas 39 a 41). Feito isso tentamos então registrar o sucesso da atualização, na tabela VERSIONAMENTO, como mostra a execução da linha 47 a 50.

Restou apenas citar o método MaisDeUmConectado. Como o requisito era executar os scripts em um banco de dados Firebird 1.5, e o mesmo não possui recursos para retornar via SQL a quantidade de usuários conectados, esse método sempre retorna falso. Contudo, nada impede que você utilize recursos avançados das versões mais recentes do Firebird para obter o resultado real.

Não serão mostrados como implementar a classe TConector para outras versões, como IBX ou DBX da versão Delphi 7 por causa da semelhança, contudo, o código fonte dos mesmo está implementado e faz parte deste framework.

Executando o teste

A primeira coisa a se fazer é incluir na seção uses as units devmedia.AtualizadorBanco e devmedia.ContectorDBXXE2.

Listagem 9. Método RetornaUltimaVersaoInstalada

```
01 function TConectorDBXXE2.RetornaUltimaVersaoInstalada: TVersao;
02 var
03  qry: TSQLDataSet;
04  obj: TVersao;
05 begin
06  if not FConexao.Connected then
07   FConexao.Open;
08
09  VerificaCriaEstruturaBasica;
10
11  qry := TSQLDataSet.Create(nil);
12  qry.SQLConnection := FConexao;
13  qry.CommandType := ctQuery;
14  qry.CommandText :='SELECT FIRST 1 * FROM VERSIONAMENTO
    WHERE APLICADA = 1 ORDER BY DATA DESC';
15  qry.Open;
16  if not qry.IsEmpty then
17  begin
18   obj := TVersao.Create;
19   obj.Id :=  qry.FieldByName('ID_VERSIONAMENTO').AsInteger;
20   obj.Numero := qry.FieldByName('NRO').AsString;
21   obj.Data := qry.FieldByName('DATA').AsDateTime;
22  end;
23  qry.Close;
24  FreeAndNil(qry);
25  Result := obj;
26 end;
```

Listagem 10. Método ExecutarScript

```
01 procedure TConectorDBXXE2.ExecutarScript(versao: TVersao);
02 var
03  comando: TStringList;
04  sentencaSQL: string;
05  s: string;
06  i: Integer;
07  Transacao: TTransactionDesc;
08 begin
09  sentencaSQL := '';
10  comando := TStringList.Create;
11  for i:= 0 to versao.Script.Count-1 do
12  begin
13   s:= versao.Script[i];
14   sentencaSQL := sentencaSQL + s;
15   if Pos(CharBlocoComando,sentencaSQL) > 0 then
16   begin
17    Delete(sentencaSQL,Pos(CharBlocoComando,sentencaSQL),1);
18    comando.Append(sentencaSQL);
19    sentencaSQL := '';
20   end
21   else
22    sentencaSQL := sentencaSQL + ' ';
23  end;
24
25  Transacao.TransactionID := 1;
26  Transacao.IsolationLevel := xilREADCOMMITTED;
27  FConexao.StartTransaction(Transacao);
28  for i:= 0 to comando.Count-1 do
29  begin
30   try
31    FConexao.ExecuteDirect(comando[i]);
32   except
33    on e: Exception do
34    begin
35     Fconexao.Rollback(Transacao);
36     raise Exception.Create(e.Message);
37    end;
38   end;
39   if i = (comando.Count -1) then
40    if FConexao.InTransaction then
41     Fconexao.Commit(Transacao);
42  end;
43  try
44   Transacao.TransactionID := 1;
45   Transacao.IsolationLevel := xilREADCOMMITTED;
46   FConexao.StartTransaction(Transacao);
47   FConexao.ExecuteDirect('insert into VERSIONAMENTO (ID_VERSIONAMENTO,
     NRO, DATA, APLICADA) ' + #13 +
48                 'values (' + IntToStr(Versao.Id)+',' + QuotedStr(Versao.Numero)+',' +
49                 QuotedStr(FormatDateTime('mm/dd/yyyy',versao.Data))+', 1)');
50   FConexao.Commit(Transacao);
51  except
52   begin
53    FConexao.Rollback(Transacao);
54    raise Exception.Create('Falha ao registrar aplicação da versão: '
     + versao.Numero);
55   end
56  end;
57 end;
```

Para testar também o framework rodando sob IBX, adiciono a unit devmedia.ConectorIBXD7 (disponível no download).

Foi criado um campo privado no formulário: conector: TConector. Ele irá armazenar qual conector o usuário escolheu utilizar. Essa escolhe acontece no momento em que se clica no botão de conexão com bancos de dados. No botão "Conectar com DBX" a conexão com o banco TESTE é feita. O detalhe fica por conta de como foi instanciado esse campo conector:

```
conector := TConectorDBXXE2.Create(SQLConnection1);
```

Instanciamos na realidade um objeto TConectorDBXXE2 e em seu construtor passamos o objeto de conexão. Isso faz com o framework utilize o dbExpress para realizar suas operações. O clique do botão "Conectar com IBX" não é muito diferente. Nele realizamos também a conexão com o banco TESTE, porém, o conector é instanciado diferentemente:

```
conector := TConectorIBX.Create(IBDatabase1);
```

Utilizamos o mesmo campo privado, mas passamos para ele uma instância de TConectorIBX.

Agora é preciso executar a atualização. A **Listagem 11** mostra esse processo do evento click do botão Atualizar.

Listagem 11. Click do botão Atualizar

```
01 procedure TForm1.btnAtualizarClick(Sender: TObject);
02 var
03   atualizador: TAtualizadorBd;
04 begin
05    memoEventos.Clear;
06   atualizador := TAtualizadorBd.Create(conector,'$');
07   atualizador.OnAplicacaoVersao := AtualizacaoAplicada;
08   atualizador.OnErroExecucaoScript := ErroAplicacaoAtualizacao;
09   atualizador.OnMaisDeUmUsuario := MaisDeUmConectado;
10   atualizador.RegistrarVersao(Versao1);
11   atualizador.RegistrarVersao(Versao1_1);
12   atualizador.AplicarAtualizacoes(atualizador.ObterListaScripts);
13
14   FreeAndNil(atualizador);
15   FreeAndNil(conector);
16 End;
```

Temos uma variável do tipo TAtualizadorBd, que é o núcleo operacional do framework. Ao instanciar esse objeto (linha 6), passamos a instância do conector selecionado anteriormente (pelos botões que realizam a conexão), mais o caractere que indicará a quebra de bloco de comandos. No caso o símbolo '$'.

Como temos três eventos disponíveis, associamos um manipulador para cada um deles (linhas 7,8 e 9). A **Listagem 12** mostra a implementação desses manipuladores. Então é preciso informar ao framework as versões disponíveis utilizando o método RegistrarVersao (linhas 10 e 11). Esse método recebe um objeto TVersao que é retornado pelos métodos Versao1 e Versao1_1, vistos na **Listagem 13**.

Com a lista de versões prontas, executamos o método AplicarAtualizacoes, que recebe como parâmetro o retorno de seu próprio método ObterListaScripts). Nesse momento o framework entra em ação, verificando a estrutura do bando de dados, executando cada script e disparando os eventos que possuem manipuladores associados. Por fim, apenas liberamos a memória utilizando FreeAndNil no objeto atualizador e conector.

Ao observar como os manipuladores dos eventos foram criados (**Listagem 12**), vemos que cada método simplesmente registra no TMemo memoEventos o que aconteceu. Mas outra possível implementação seria exibir uma caixa de mensagens exibindo o erro ou até mesmo registrar em um arquivo todo o histórico do processo.

Listagem 12. Manipuladores dos eventos

```
procedure TForm1.AtualizacaoAplicada(Sender: TObject; Versao: TVersao);
begin
  memoEventos.Lines.Append('Atualização aplicada: '+ Versao.Numero);
  memoEventos.Lines.Append('Detalhes: '+ Versao.Detalhes);
  memoEventos.Lines.Append('----------------------------------------');
end;

procedure TForm1.ErroAplicacaoAtualizacao(Sender: TObject;
  Versao: TVersao; Excecao: Exception);
var
  i: Integer;
begin
  memoEventos.Lines.Append('Falha ao aplicar atualização: ' + Versao.Numero);
  memoEventos.Lines.Append('Mensagem da exceção: ');
  memoEventos.Lines.Append(Excecao.Message);
  memoEventos.Lines.Append('Script:');
  for i:= 0 to Versao.Script.Count-1 do
    memoEventos.Lines.Append(Versao.Script[i]);
    memoEventos.Lines.Append('----------------------------------------');
end;

procedure TForm1.MaisDeUmConectado(Sender: TObject);
begin
  memoEventos.Lines.Append('Mais de um usuário conectado');
  memoEventos.Lines.Append('----------------------------------------');
end;
```

A criação das versões também é bem simples. Na **Listagem 13** vemos que as sentenças SQL são concatenadas e encerram com o caractere definido anteriormente, o '$'. Por exemplo, no método Versao1 temos a definição de um generator e uma tabela, e eles são separados por esse caractere.

Nota do DevMan 1

MDO, ou Mercury Database Objects é uma suíte de acesso a dados gratuita, que oferece acesso nativo ao Firebird. Alías, é compatível que todas versões do Firebird. Um detalhe muito interessante sobre essa suíte está no fato de que seu código fonte é baseado nas bibliotecas FreeIBComponents e no IBX versão 4.3 Pode-se dizer que o MDO é compatível com o IBX, facilitando a migração para quem deseja ter acesso aos últimos recursos do Firebird e não pode por estar no IBX ainda. Entre seus recursos está a disponibilização de componentes que dão suporte a operações administrativas como backup e restore e até mesmo serviço de validação.

Listagem 13. Criação das versões

```
function TForm1.Versao1: TVersao;
var
 versao: TVersao;
begin
 versao := TVersao.Create;
 versao.Id := 1;
 versao.Numero := '1';
 versao.Data := Now;
 versao.Detalhes := 'Teste de criação de script geral';
 versao.Script.Add('CREATE GENERATOR GEN_TABELA1;');
 versao.Script.Add('$');
 versao.Script.Add('CREATE TABLE TABELA1 (');
 versao.Script.Add('ID_TABELA1  INTEGER NOT NULL,');
 versao.Script.Add('CAMPO1     VARCHAR(100),');
 versao.Script.Add('CAMPO2     VARCHAR(100)');
 versao.Script.Add(');$');
 Result := versao;
end;

function TForm1.Versao1_1: TVersao;
var
 versao: TVersao;
begin
 versao := TVersao.Create;
 versao.Id := 2;
 versao.Numero := '1.1';
 versao.Data := Now;
 versao.Detalhes := 'Teste de criação trigger e chave primaria';
 versao.Script.Add('ALTER TABLE TABELA1 ADD CONSTRAINT PK_TABELA1
  PRIMARY KEY (ID_TABELA1);');
 versao.Script.Add('$');
 versao.Script.Add('CREATE TRIGGER TABELA1_BI FOR TABELA1');
 versao.Script.Add('ACTIVE BEFORE INSERT POSITION 0');
 versao.Script.Add('as');
 versao.Script.Add('begin');
 versao.Script.Add('if (new.id_tabela1 is null) then');
 versao.Script.Add('  new.id_tabela1 = gen_id(gen_tabela1,1);');
 versao.Script.Add('end');
 versao.Script.Add('$');
 Result := versao;
end;
```

Conclusão

É importante manter a evolução dos negócios de uma empresa, mantendo sua boa impressão com seus clientes, sejam eles 10 ou 100. Criar pequenos frameworks que automatizam rotinas simples, como esta, ajudam a manter a ideia de agilidade no atendimento. Esse framework pode ser estendido para ir além do que foi feito. O foco dele foi o Firebird 1.5, mas nada impede que você faça alguns ajustes, reestruture a arquitetura para ser utilizado com qualquer banco. Outro ponto a ser evidenciado é a utilização da orientação a objetos. Veja que não foi preciso criar nenhum "if" para fazer com o framework ora funcione com IBX ora com dbExpress. A herança, polimorfismo e acima de tudo a composição permitiram um pequeno software ser bem flexível. Se você utiliza MDO (ver **Nota do DevMan 1**) para conectar com seu banco de dados, basta adicionar uma nova unit, descender uma nova classe que herde de TConector, implementar seus métodos com os recursos do MDO e na hora de utilizar, incluí-la na seção uses de seu projeto. Com isso atendemos o último requisito exigido: simples integração.

Paulo Quicoli
pauloquicoli@gmail.com
Editor da revista ClubeDelphi e editor técnico .Net Magazine. Formado em processamento de dados pela FATEC-TQ. Atua como analista programador de sistemas da Siplan Control-M (www.siplancontrolm.com.br) unidade Jaboticabal, e professor da FATEC, campus Taquaritinga.
Blog: http://quicoli.worpress.com

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

JSON – Serializando Objetos

Transfira objetos em aplicações multicamadas usando JSON

Muitos desenvolvedores têm dúvidas quanto à utilização da orientação a objetos em Delphi. Apesar de a linguagem ter suporte a este paradigma, são poucos os desenvolvedores que o aplicam corretamente em suas aplicações. Isso acontece muito pelo Delphi ser uma ferramenta RAD, o que faz com que uma aplicação multicamadas possa ser desenvolvida muito rapidamente apenas ligando os componentes SQLDataSet, DataSetProvider e ClientDataSet.

Porém, ao utilizar o Delphi somente desta maneira não utilizamos o máximo de reaproveitamento de código para projetos futuros e são deixados de fora excelentes recursos que a Linguagem e IDE vem adicionando. O Delphi evoluiu muito e trouxe para os desenvolvedores vários novos recursos que permitem a aplicação da orientação a objetos da melhor forma, como JSON, Generics, DBX4, Datasnap, RTTI, Anonymous Methods e muitos outros. Vamos abordar alguns desses recursos.

Generics

Generics é um recurso que foi incorporado na versão 2009 do Delphi. Como o nome diz, este recurso permite que sejam criadas estruturas genéricas. Na orientação a objetos podemos definir o tipo de retorno de uma lista de objetos e somente serão permitidas inserções de objetos daquele tipo, do contrário ocorrerá um erro ainda em tempo de compilação, assim, evitando erros em tempo de execução do tipo Invalid Type Casting, muito comum para quem trabalhava com listas de objetos em versões anteriores a 2009 em Delphi.

Representação de objetos em JSON

JSON (JavaScript Object Notation) é um formato leve para troca de dados. É simples de se entender e fácil para as máquinas manipularem. Devido a esta simplicidade, tem se tornado uma boa alternativa ao uso de XML que possui tags diversas e regras mais complexas.

Resumo DevMan

JSON – Serializando Objetos:

Neste artigo veremos como utilizar o formato JSON para troca de dados entre camadas em Delphi. Usaremos Generics, recurso que permite flexibilizar e dar segurança ao código através da tipagem de métodos e listas. Aprenderemos a usar as classes que o dbExpress 4 nos oferece e que facilitam muito o desenvolvimento orientado a objetos. Aplicaremos o padrão DAO para centralizar o acesso a dados em classes separadas das classes de domínio. Através do Datasnap vamos criar um Servidor de Aplicações e disponibilizar métodos para as aplicações clientes. Aplicaremos alguns padrões de projeto que irão tornar o projeto ainda melhor, como Singleton e Factory. Aprenderemos o conceito de Injeção de Dependência que irá tornar nossas classes ainda mais desacopladas. No final teremos uma aplicação cliente feita com o Firemonkey que irá fazer uso do servidor de aplicação.

Em que situação o tema é útil:

Quando temos uma aplicação multicamadas, transferimos informações entre uma aplicação servidora e uma cliente, o JSON é útil justamente para otimizar essa transferência, diminuindo o tamanho das informações que são enviadas/recebidas. Quanto à arquitetura e recursos utilizados (generics e padrões de projeto) são úteis quando queremos flexibilidade, centralização das regras de negócio e independência de linguagem das aplicações clientes.

O JSON tem um formato texto e é completamente independente de linguagem de programação, tornando-se ideal para trocas de dados entre as mais diversas plataformas e linguagens.

Um objeto em JSON é constituído de um conjunto de pares nome/valor. Todo objeto começa com { (abertura de chave) e termina com } (fechadura de chave). Cada nome é seguido por : (dois pontos) e os pares nome/valor são seguidos por , (vírgula), como mostra a **Figura 1**.

Podemos representar também um Array de Objetos em JSON. Ele começa com [(abertura de colchete) e termina com] (fechadura de colchete). Os objetos são separados por vírgula (**Figura 2**).

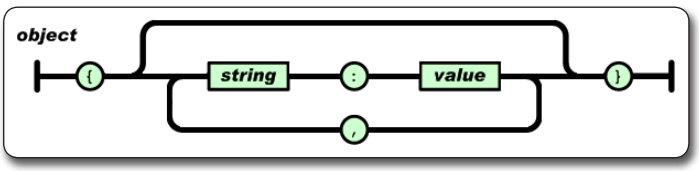


Figura 1. Representação de um objeto JSON

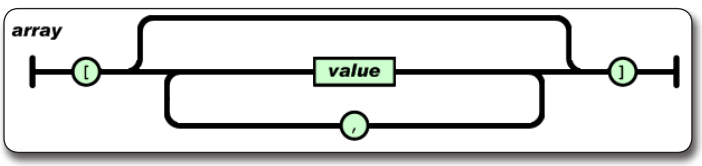


Figura 2. Representação de um array de objetos JSON

dbExpress 4

O dbExpress, que surgiu na versão 6 do Delphi, passou por uma grande mudança na versão 4 lançada no Delphi 2007. Foi reescrito totalmente em Delphi, sendo 100% orientado a objetos e possui diversas classes que ajudam muito os desenvolvedores que utilizam este paradigma. As principais são, a classe TDBXConnection que tem a função de gerenciar a conexão com o Banco de Dados. Ela também oferece métodos para a criação de comandos, transações e metadados. Já a classe TDBXCommand é responsável por executar comandos SQL e StoredProcedures. Os parâmetros podem ser acessados através da propriedade Parameters. Outros tipos de comandos podem ser executados pela classe, estes estão disponíveis através da propriedade TDBXCommand.CommandType. A classe TDBXTransaction é responsável por gerenciar as transações. Um transação pode ser criada usando o método BeginTransaction da classe TDBXConnection. Alguns outros métodos disponíveis são CommitFreeAndNil, RollbackAndFreeAndNil. Podemos definir também um nível de isolamento das transações através da classe TDBXIsolation. Outra é a TDBXReader que oferece um leitor unidirecional para um conjunto de dados. Quem provê estes dados é a classe TDBXCommand através do método TDBXCommand.ExecuteQuery. As linhas podem ser percorridas através do método TDBXReader.Next e os dados coletados através da propriedade TDBXReader.Value, sendo pelo nome ou pela posição do campo na matriz de valores.

Datasnap

O Datasnap vem trazendo várias novidades nas últimas versões do Delphi e em 2009 passou por uma reestruturação. Servidores Datasnap podem ser desenvolvidos tanto em Delphi quanto C++ Builder, porém a grande vantagem está na aplicação cliente, que pode ser desenvolvida em qualquer linguagem de programação que tenha suporte a JSON. A **Tabela 1** exibe os principais componentes e sua função dentro do framework.

Componente	Função
TDSServer	Transforma a aplicação em um Servidor de Aplicações.
TDSTCPServerTransport	Responsável pela comunicação via TCP-IP.
TDSServerClass	Classe responsável por publicar os métodos para o Cliente.
TDSHttpService	Responsável pela comunicação via protocolo HTTP.

Tabela 1. Componentes do Datasnap

Interfaces

Interfaces são contratos estabelecidos para que as classes que desejam implementar a interface, o sigam. Elas possuem apenas a definição dos métodos e nunca a implementação. Uma interface só expõe o que a classe que deseja implementá-la deve fazer, mas não como deve ser feito.

Uma regra de ouro na programação orientada o objetos é a de que devemos sempre programar voltado para interface de nossas classes e nunca para a implementação, pois mudanças na interface afetam outras classes que se utilizam desta, já mudanças na implementação podem ser feitas a qualquer momento sem que sejam necessárias mudanças nas classes dependentes desta.

Padrão DAO

DAO (Data Access Object) ou objeto de acesso a dados é um padrão de projeto que visa separar as regras de negócio das regras de acesso a dados. Surgiu em aplicações J2EE do Java, mas que pode ser aplicada a qualquer linguagem de programação orientada a objetos. Este padrão permite mudar a forma de persistência sem que isso influencie em nada na lógica de negócio.

Classes DAO são responsáveis por trocar informações com o SGBD e fornecer operações CRUD e de pesquisas, ela deve ser capaz de buscar dados no banco e transformá-los em objetos, também deverá receber os objetos, converter em instruções SQL e mandar para o bando de dados.

Se este padrão for aplicado corretamente ele vai abstrair completamente o modo de busca e gravação dos dados, tornando isso transparente para aplicação e facilitando muito na hora da manutenção na aplicação ou, na migração para um banco de dados diferente, isso porque tem-se um ponto único de acesso a dados.

Nota

Persistência aqui diz respeito a salvar um objeto que está em memória para algum meio físico, seja ele um arquivo XML, JSON e mais comumente utilizado, um banco de dados relacional.

Injeção de Dependência

Dependency Injection (DI) ou injeção de dependência é um padrão aplicado em software para manter um baixo acoplamento entre diferentes módulos do sistema. As dependências de uma classe não devem ser criadas internamente a ela e sim ser "injetadas", ou fornecidas.

Esses assuntos serão mostrados agora na prática, fornecendo um melhor entendimento sobre seu funcionamento e implementação.

Tutorial

Servidor de aplicação

Vamos criar nosso servidor usando o Wizard do Delphi (File>New>Other>DataSnap Server) para isso, marcando algumas opções, como mostrado na **Figura 3**.

A interface do servidor de aplicação é bem simples e contém apenas dois botões, um para iniciar e outro para parar o servidor, veja a **Figura 4**. O códigos dos botões e do evento OnShow do Formulário estão na **Listagem 1**.

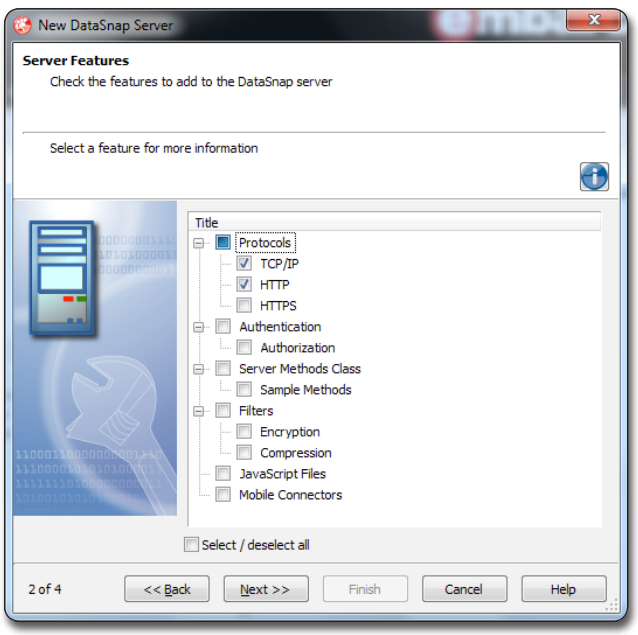


Figura 3. Wizard de criação do servidor de aplicações

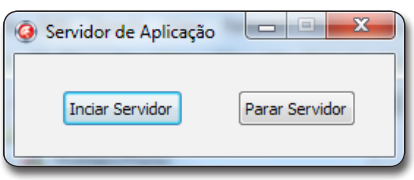


Figura 4. Interface do servidor

Ao clicar no iniciar botão iniciar, o IniciarClick é disparado. Nele verificamos se o servidor já está em execução (linha 3), não estando, simplesmente o iniciamos executando seu método Start (linha 5). E para indicar isso visualmente ao usuário, desabilitamos o botão iniciar e habilitamos o parar. Já no clique no botão btnParar realizamos o processo contrário (linhas 12 à 14). E para que a tela apareça ao usuário em conformidade com o comportamento, no evento FormShow (linha 16) habilitamos os botões de acordo com o estado do servidor, lendo sua propriedade Started, que retorna True caso o servidor já esteja em execução.

Classe Modelo

A **Listagem 2** apresenta a classe TConta. Esta classe vai representar uma conta bancária na aplicação. Observe que não precisamos fazer nenhuma herança em nossa classe modelo para que possamos trabalhar com ela, é um PODO básico (ver **Nota do DevMan 1**). A ideia é justamente esta, ter as classes desacopladas das demais.

Listagem 1. Código dos botões e do evento OnShow

```
01 procedure TfrmMain.btnIniciarClick(Sender: TObject);
02 begin
03   if not ServerContainer.DSServer.Started then
04   begin
05     ServerContainer.DSServer.Start;
06     btnIniciar.Enabled := False;
07     btnParar.Enabled := True;
08   end;
09 end;
10 procedure TfrmMain.btnPararClick(Sender: TObject);
11 begin
12   ServerContainer.DSServer.Stop;
13   btnIniciar.Enabled := True;
14   btnParar.Enabled := False;
15 end;
16 procedure TfrmMain.FormShow(Sender: TObject);
17 begin
18   btnIniciar.Enabled := not ServerContainer.DSServer.Started;
19   btnParar.Enabled := ServerContainer.DSServer.Started;
20 end;
```

Listagem 2. Classe TConta

```
01 TConta = class
02 private
03   FId: Integer;
04   FNumero: string;
05   FCliente: string;
06   FSaldo: Double;
07 public
08   property Id: Integer read FId write FId;
09   property Numero: string read FNumero write FNumero;
10   property Cliente: string read FCliente write FCliente;
11   property Saldo: Double read FSaldo write FSaldo;
12 end;
```

Nota do DevMan 1

PODO (Plain Old Delphi Object) – é um objeto simples que não precisa herdar nem implementar nada.

Tabela para persistência

Para o exemplo foi usado o banco de dados gratuito Firebird, versão 2.5. Para o exemplo pode ser criado um novo banco de dados em um local de fácil acesso e nele será criada a tabela CONTAS. O script de criação da tabela pode ser visto na **Listagem 3**. Também foi criado um Generator e Trigger para que o Firebird incremente automaticamente a chave primária ID_CONTA da tabela CONTAS. Não serão dados maiores detalhes sobre esses temas, pois não é o foco deste artigo.

Listagem 3. Script da tabela Conta

```
CREATE GENERATOR GEN_CONTAS_ID;

CREATE TABLE CONTAS (
   ID_CONTA   INTEGER NOT NULL,
   NUMERO     VARCHAR(20) NOT NULL,
   CLIENTE    VARCHAR(50) NOT NULL,
   SALDO      DECIMAL(10,2) NOT NULL
);

ALTER TABLE CONTAS ADD CONSTRAINT PK_CONTAS PRIMARY KEY (ID_CONTA);
ALTER TABLE CONTAS ADD CONSTRAINT UNQ1_CONTAS UNIQUE (NUMERO);

CREATE OR ALTER TRIGGER CONTAS_BI FOR CONTAS
ACTIVE BEFORE INSERT POSITION 0
as
begin
 if (new.id_conta is null) then
   new.id_conta = gen_id(gen_contas_id,1);
end
```

Conexão dbExpress

Foi criada a classe TDBConnection. Ela tem o papel de encapsular todo o trabalho referente ao dbExpress e os objetos necessários, fornecendo propriedades e métodos que poderão ser reutilizados em vários outros projetos. Este é um dos pilares da Orientação a Objetos, tornar classes reaproveitáveis.

A **Listagem 4** apresenta a interface da classe TDBConnection de conexão utilizando o dbExpress. Veja que a classe possui a diretiva Sealed que indica que não devemos criar outras classes herdando desta. Esta classe implementa o padrão de projeto Singleton, isto porque é aconselhado ter apenas uma instância de conexão com o banco em nossa aplicação, já que estabelecer uma conexão é uma operação custosa. Por este motivo temos como único meio de obter uma instância da classe, o método GetInstance. Esta instância será armazenada internamente através variável de classe FInstance. Outro detalhe importante é o fato de movermos o construtor para a seção Private, assim sendo inacessível por outras classes.

Na cláusula Uses temos Data.DBXFirebird e Data.DBXDB2 (ver **Nota do DevMan 2**), fazendo isto nossa classe está apta a usar os drivers tanto do Firebird quando do DB2. Se quiséssemos usar outros bancos de dados, bastaria adicionar a Unit referente ao banco de dados desejado.

Nota do DevMan 2

Unit Scopes Names – novo recurso de nomeação de Units presente desde o Delphi XE 2, agora é possível criar prefixos nas Units. A partir do Delphi XE2 algumas units do Delphi foram prefixadas, como VCL, RTL, FMX. Por exemplo, agora temos Vcl.Dialogs ao invés de Dialogs.

Na **Listagem 5** temos a implementação do método GetInstance, veja que ele verifica se já não existe uma instância da classe na aplicação fazendo uso da função Assigned, se existir ele devolve a existente, senão ele chama o Create que é privado, mas podemos acessá-lo internamente a nossa classe.

Na **Listagem 6** temos a implementação do construtor Create. Ele utiliza o método GetConnectionFactory.GetConnection da classe TDBXConnectionFactory para criar uma instância TDBXConnection, que representa a conexão com o banco de dados, em seguida cria o objeto TDBXCommand através do método CreateCommand, para que possamos executar instruções SQL através da propriedade Command.

Listagem 4. Declaração de TDBConnection

```
01 unit Server.DBConnection;
02 interface
03 uses Data.DBXCommon, Data.DBXFirebird, Data.DBXDB2;
04 type
05 TDBConnection = class sealed
06  private
07   FConnection: TDBXConnection;
08   FTransaction: TDBXTransaction;
09   FCommand: TDBXCommand;
10   FReader: TDBXReader;
11   class var FInstance: TDBConnection;
12   procedure FreeObjects;
13   constructor Create;
14 public
15   class function GetInstance: TDBConnection;
16   property Command: TDBXCommand read FCommand;
17   property Reader: TDBXReader read FReader write FReader;
18   property Connection: TDBXConnection read FConnection;
19   procedure BeginTransaction;
20   procedure Commit;
21   procedure Roolback;
22  end;
```

Listagem 5. Implementação de GetInstance

```
01 class function TDBConnection.GetInstance: TDBConnection;
02 begin
03  if not Assigned(FInstance) then
04   FInstance := Create;
05  Result := FInstance;
06 end;
```

Listagem 6. Construtor Create

```
01 constructor TDBConnection.Create;
02 begin
03  FConnection := TDBXConnectionFactory.GetConnectionFactory.
    GetConnection
04   ('DB_BANCO', 'SYSDBA', 'masterkey');
05  FCommand := FConnection.CreateCommand;
06 end;
```

Em seguida temos os métodos referente às transações, são eles BeginTransaction, Commit, Roolback. O método BeginTransaction usa a classe TDBXConnection para iniciar uma transação e atribui esta transação ao objeto FTransaction. O métodos Commit e Rollback usam novamente o objeto de TDBXConnection para fazer CommitFreeAndNil e RollbackFreeAndNil que além de fazer as operações de Commit e Rollback também liberam o objeto que representa a transação da memória. Veja **Listagem 7**.

Para finalizar temos na **Listagem 8** a seção Finalization da Unit a chamada do método FreeObjects que vai liberar todos os recursos da memória antes da aplicação ser finalizada.

Listagem 7. Métodos referentes às transações

```
01 procedure TDBConnection.BeginTransaction;
02 begin
03   FTransaction := FConnection.BeginTransaction;
04 end;
05 procedure TDBConnection.Commit;
06 begin
07   FConnection.CommitFreeAndNil(FTransaction);
08 end;
09 procedure TDBConnection.Roolback;
10 begin
11   FConnection.RollbackFreeAndNil(FTransaction);
12 end;
```

Listagem 8. Seção Finalization e método FreeObjects

```
01 procedure TDBConnection.FreeObjects;
02 begin
03   FReader.Free;
04   FCommand.Free;
05   FConnection.Free;
06   FTransaction.Free;
07   FInstance.Free;
08 end;
09 initialization
10 finalization
11   TDBConnection.FInstance.FreeObjects;
```

DAO

Como dito anteriormente, termos classes DAO para trabalhar com a persistência de dados. Normalmente teremos uma classe DAO para cada classe de negócio (TConta) ou uma para cada módulo do sistema onde temos classes fortemente relacionadas.

O primeiro passo é criarmos uma interface, que no nosso caso será IContaDao, conforme a **Listagem 9**. Outras classes vão interagir somente com a interface, nunca com a implementação das classes DAO.

Temos o método Save que será responsável por fazer um Insert no banco de dados, o método Update que irá fazer uma atualização em um registro, o Delete que irá apagar um registro e dois métodos de recuperação de objetos, o FindAll que irá retornar uma lista de todas as contas e o método FindById que irá retornar um único objeto a partir do identificador passado como parâmetro.

Agora basta criarmos uma classe que implemente esta interface, como por exemplo a **Listagem 10**. Observe que a implementação terá um objeto FDBConnection do tipo TDBConnection, que é a nossa classe de conexão com o banco de dados através do dbExpress vista anteriormente.

Atente para o construtor desta classe, veja que ele recebe por parâmetro o objeto de conexão pronto, ou seja, nossa classe DAO não será responsável por criar este objeto e sim receberá já instanciado por outra classe. Veremos a seguir o porquê disso.

Apesar de termos a interface DAO não é interessante chamar diretamente o Create da classe TContaDao quando quisermos fazer a persistência da classe TConta, isto porque será criada uma dependência para a implementação da classe TContaDao. O ideal aqui é usar o padrão de projeto Factory (fábrica) para que seja criado uma instância da classe e para que seja injetada a dependência DBConnection. Veja na **Listagem 11** a interface e implementação desta fábrica.

Listagem 9. Interface IContaDao

```
01  IContaDao = interface
02    ['{3A6F9C5E-67EC-4483-BE78-FC425A2AF0F9}']
03    procedure Save(Conta: TConta);
04    procedure Update(Conta: TConta);
05    procedure Delete(Conta: TConta);
06    function FindAll: TList<TConta>;
07    function FindById(const Id: Integer): TConta;
08  end;
```

Listagem 10. Implementação TContaDao

```
01  TContaDao = class(TInterfacedObject, IContaDao)
02  strict private
03    FDBConnection: TDBConnection;
04  public
05    constructor Create(DBConnection: TDBConnection);
06    procedure Save(Conta: TConta);
07    procedure Update(Conta: TConta);
08    procedure Delete(Conta: TConta);
09    function FindAll: TList<TConta>;
10    function FindById(const Id: Integer): TConta;
11  end;
```

Listagem 11. Factory CreateContaDao

```
01  TFactoryConta = class
02    class function CreateContaDao: TContaDao;
03  end;
04
05      class function TFactoryConta.CreateContaDao: TContaDao;
06      begin
07        Result := TContaDao.Create(TDBConnection.GetInstance);
08      end;
```

Observe atentamente que a fábrica injeta o objeto de conexão na nossa classe DAO. Este padrão serve justamente para isso, criar família de objetos complexos, no momento que nossa classe TContaDao ou TDBConnection sofrerem alguma modificação, é somente neste ponto que são conhecidas as implementações desta.

A seguir temos as implementações dos métodos da nossa classe concreta DAO. Começamos pelo método Save conforme a **Listagem 12**.

Inicialmente criamos uma constante (linha 3) com o SQL de insert e com os parâmetros definidos com "?". Essa é também uma novidade do dbExpress 4. Essa sentença SQL é passada à conexão na linha 7. Nas linhas 9 à 11, fazemos as atribuições necessárias dos parâmetros e então é executado o Insert no banco de dados através do método ExecuteUpdate (linha 14) da classe TDBXCommand. Se estiver tudo certo é efetuado o commit senão é feito um rollback.

Os métodos Update e Delete seguem as mesmas regras apenas mudando os parâmetros e instrução SQL. Veja na **Listagem 13**.

Listagem 12. Método Save

```
01 procedure TContaDao.Save(Conta: TConta);
02 const
03   SQL_INSERT = 'insert into CONTAS (NUMERO, CLIENTE, SALDO) values (?, ?, ?)';
04 begin
05   with FDBConnection do
06   begin
07     Command.Text := SQL_INSERT;
08     Command.Prepare;
09     Command.Parameters[0].Value.AsString := Conta.Numero;
10     Command.Parameters[1].Value.AsString := Conta.Cliente;
11     Command.Parameters[2].Value.AsDouble := Conta.Saldo;
12     BeginTransaction;
13     try
14       Command.ExecuteUpdate;
15       Commit;
16     except
17       Roolback;
18     end;
19   end;
20 end;
```

Listagem 13. Métodos Update e Delete

```
01 procedure TContaDao.Update(Conta: TConta);
02 const
03   SQL_UPDATE = 'update CONTAS set NUMERO = ?, CLIENTE = ?, SALDO = ?';
04 begin
05   with FDBConnection do
06   begin
07     Command.Text := SQL_UPDATE;
08     Command.Prepare;
09     Command.Parameters[0].Value.AsString := Conta.Numero;
10     Command.Parameters[1].Value.AsString := Conta.Cliente;
11     Command.Parameters[2].Value.AsDouble := Conta.Saldo;
12     Command.Parameters[3].Value.AsInt32 := Conta.Id;
13     BeginTransaction;
14     try
15       Command.ExecuteUpdate;
16       Commit;
17     except
18       Roolback;
19     end;
20   end;
21 end;
22 procedure TContaDao.Delete(Conta: TConta);
23 const
24   SQL_DELETE = 'delete from CONTAS where ID_CONTA = ?';
25 begin
26   with FDBConnection do
27   begin
28     Command.Text := SQL_DELETE;
29     Command.Prepare;
30     Command.Parameters[0].Value.AsInt32 := Conta.Id;
31     BeginTransaction;
32     try
33       Command.ExecuteUpdate;
34       Commit;
35     except
36       Roolback;
37     end;
38   end;
39 end;
```

Temos algumas mudanças nos métodos FindById e FindAll, conforme a **Listagem 14**. Agora iremos popular um objeto TConta no objeto FindById, e criar uma lista de objetos no método FindAll, para então devolvermos para a aplicação cliente. Para isso usaremos a propriedade Reader da classe TDBXReader que receberá os dados de um comando SQL executado por TDBXCommand.

Listagem 14. Métodos FindById e FindAll

```
01 function TContaDao.FindAll: TList<TConta>;
02 const
03   SQL_FIND_ALL = 'select ID_CONTA, NUMERO, CLIENTE, SALDO from CONTAS';
04 var
05   Conta: TConta;
06 begin
07   Result := TList<TConta>.Create;
08   with FDBConnection do
09   begin
10     Command.Text := SQL_FIND_ALL;
11     Reader := Command.ExecuteQuery;
12     while Reader.Next do
13     begin
14       Conta := TConta.Create;
15       Conta.Id := Reader.Value['ID_CONTA'].AsInt32;
16       Conta.Numero := Reader.Value['NUMERO'].AsString;
17       Conta.Cliente := Reader.Value['CLIENTE'].AsString;
18       Conta.Saldo := Reader.Value['SALDO'].AsDouble;
19       Result.Add(Conta);
20     end;
21   end;
22 end;
23 function TContaDao.FindById(const Id: Integer): TConta;
24 const
25   SQL_FIND_ID = 'select ID_CONTA, NUMERO, CLIENTE, SALDO from CONTAS
       where ID_CONTA = ?';
26 var
27   Conta: TConta;
28 begin
29   with FDBConnection do
30   begin
31     Command.Text := SQL_FIND_ID;
32     Command.Prepare;
33     Command.Parameters[0].Value.AsInt32 := Id;
34     Reader := Command.ExecuteQuery;
35     while Reader.Next do
36     begin
37       Conta := TConta.Create;
38       Conta.Id := Reader.Value['ID_CONTA'].AsInt32;
39       Conta.Numero := Reader.Value['NUMERO'].AsString;
40       Conta.Cliente := Reader.Value['CLIENTE'].AsString;
41       Conta.Saldo := Reader.Value['SALDO'].AsDouble;
42       Result := Conta;
43     end;
44   end;
45 end;
```

JSON Util

Para as conversões JSON para Objeto, Objeto para JSON, Array de Objetos para Array JSON e Array JSON para Array de Objetos criaremos uma classe para este fim.

Será uma classe com métodos estáticos, já que esta é uma classe utilitária e não faz muito sentido termos instância desta classe em nossa aplicação e sim usar seus métodos para as conversões necessárias. Veja na **Listagem 15** a declaração.

Listagem 15. Declaração da Classe para conversões JSON

```
01 unit Common.JsonUtil;
02 interface
03 uses
04  Data.DBXJSON, Data.DBXJSONReflect, Generics.Collections;
05 type
06  TJsonUtil = class
07  private
08   class var JSONMarshal: TJSONMarshal;
09   class var JSONUnMarshal: TJSONUnMarshal;
10  public
11   class function ObjectToJSON<T: class>(AObject: T): TJSONValue;
12   class function JSONToObject<T: class>(AJSON: TJSONValue): T;
13   class function ListToJSONArray<T: class>(AList: TList<T>): TJSONArray;
14   class function JSONArrayToList<T: class>(AJSONArray: TJSONArray): TList<T>;
15  end;
```

Primeiro de tudo são declarados (linhas 8 e 9) dois objetos do tipo TJSONMarshal e TJSONUnMarshal, estes são responsáveis por fazer o trabalho pesado, converter de Objeto para JSON no primeiro e de JSON para Objeto no segundo.

Em seguida entra em ação as Constraints do Generics (linha 11, 12, 13 e 14), veja que antes de criar o parâmetro definimos o tipo de dados que será suportado e em seguida passamos o objeto do mesmo tipo como parâmetro. No exemplo estamos dizendo que irá ser suportada qualquer classe como parâmetro já que o T é do tipo Class.

No primeiro método recebemos um objeto por parâmetro e devolvemos um TJSONValue, o segundo temos o oposto é passado um TJSONValue e retornado um objeto. Veja como é fantástico utilizar Generics neste momento, antes de passar o parâmetro informamos o tipo de retorno que desejamos, isso faz com que a classe TJSONUtil se torne Genérica, sirva para conversões de qualquer tipo de objetos para JSON e vice-versa. A **Listagem 16** mostra sua implementação.

Listagem 16. Conversões de Objetos para JSON e JSON para Objeto

```
01 class function TJsonUtil.ObjectToJSON<T>(AObject: T): TJSONValue;
02 begin
03  if Assigned(AObject) then
04  begin
05   if not Assigned(JSONMarshal) then
06    JSONMarshal := TJSONMarshal.Create;
07    Result := JSONMarshal.Marshal(AObject);
08  end
09  else
10   Exit(TJSONNull.Create);
11 end;
12 class function TJsonUtil.JSONToObject<T>(AJSON: TJSONValue): T;
13 begin
14  if AJSON is TJSONNull then
15   Exit(nil);
16  if not Assigned(JSONUnMarshal) then
17   JSONUnMarshal := TJSONUnMarshal.Create;
18  Result := T(JSONUnMarshal.UnMarshal(AJSON));
19 end;
```

Seguindo com nossa classe TJSONUtil, precisaremos de métodos genéricos para trabalhar com listas de objetos e é claro que temos suporte para isso no Delphi. O primeiro deles converte uma lista de objetos para um TJSONArray. São percorridos os itens da lista e através do método AddElement conseguimos adicionar um novo TJSONValue em nosso TJSONArray. Observe que para não haver repetição de código são chamados os métodos já existentes em nossa classe para cada item da lista.

No segundo método temos a conversão de um TJSONArray para uma Lista de objetos, cujo tipo também é passado e recebemos exatamente a lista de objetos que desejamos. Neste caso fazemos um For de 0 até a propriedade Size -1 do ArrayList e usamos o método Get(i) para recuperar um TJSONValue desta JSONArray. Veja a implementação destes métodos na **Listagem 17**.

Listagem 17. ArrayList para TList<T> e TList<T> para ArrayList

```
01 class function TJsonUtil.ListToJSONArray<T>(AList: TList<T>): TJSONArray;
02 var i: Integer;
03 begin
04  Result := TJSONArray.Create;
05  for i := 0 to AList.Count - 1 do
06   Result.AddElement(Self.ObjectToJSON(AList[i]));
07 end;
08 class function TJsonUtil.JSONArrayToList<T>(AJSONArray: TJSONArray): TList<T>;
09 var i: Integer;
10 begin
11  Result := TList<T>.Create;
12  for i := 0 to AJSONArray.Size - 1 do
13   Result.Add(Self.JSONToObject<T>(AJSONArray.Get(i)));
14 end;
```

Publicando os métodos

Outro passo necessário no lado do servidor é criar uma classe responsável por publicar os métodos no servidor e fazer as conversões necessárias. Veja a declaração desta classe na **Listagem 18**.

O primeiro detalhe a ser comentado é o da declaração desta classe estar contornada pelas diretivas de compilação {$METHODINFO ON} e {$METHODINFO OFF}, além de herdar da classe TPersistent. Isto é necessário para que o compilador consiga extrair informações da classe via RTTI.

Veja que não foi necessário criar um ServerModule para publicar nossos métodos, com esta estrutura nossos métodos estarão disponíveis para serem publicados. Observe também que usaremos apenas a interface da nossa classe DAO e usaremos a classe TFactoryConta para criar as instâncias necessárias para os métodos.

O método Save vai receber da aplicação Cliente um TJSONValue como parâmetro, depois somente chama o método Save da interface IContaDao, lembrando de fazer a conversão para Objetos novamente para que possa ser persistido. O processo é semelhante nos métodos Update e Delete.

A mudança ocorre nos métodos FindById e FindAll, porque neste caso temos que devolver para a aplicação cliente um resultado.

Listagem 18. Unit Server.TSMConta

```
01 unit Server.SMConta;
02 interface
03 uses
04   System.Classes, Data.DBXJSON, Server.ContaDao;
05 type
06   {$METHODINFO ON}
07   TSMConta = class(TPersistent)
08   private
09     FContaDao: IContaDao;
10   public
11     procedure Save(AObject: TJSONValue);
12     procedure Delete(AObject: TJSONValue);
13     procedure Update(AObject: TJSONValue);
14     function FindAll: TJSONArray;
15     function FindById(const Id: Integer): TJSONValue;
16   end;
17   {$METHODINFO OFF}
18 implementation
19 uses
20   Common.JsonUtil, Common.Conta;
21 { TSMConta }
22 procedure TSMConta.Delete(AObject: TJSONValue);
23 begin
24   FContaDao := TFactoryConta.CreateContaDao;
25   FContaDao.Delete(TJsonUtil.JSONToObject<TConta>(AObject));
26 end;
27 function TSMConta.FindAll: TJSONArray;
28 begin
29   Result := TJsonUtil.ListToJSONArray<TConta>
       (TFactoryConta.CreateContaDao.FindAll);
30 end;
31 function TSMConta.FindById(const Id: Integer): TJSONValue;
32 begin
33   Result := TJsonUtil.ObjectToJSON<TConta>
       (TFactoryConta.CreateContaDao.FindById(Id));
34 end;
35 procedure TSMConta.Save(AObject: TJSONValue);
36 begin
37   FContaDao := TFactoryConta.CreateContaDao;
38   FContaDao.Save(TJSONUtil.JSONToObject<TConta>(AObject));
39 end;
40 procedure TSMConta.Update(AObject: TJSONValue);
41 begin
42   FContaDao := TFactoryConta.CreateContaDao;
43   FContaDao.Update(TJsonUtil.JSONToObject<TConta>(AObject));
44 end;
45 end.
```

No FindById conseguimos em uma linha resolver o problema, neste caso fazemos o contrário, chamaremos o método ObjectToJSON, definimos o tipo de retorno, chamamos o Create da Factory e chamamos diretamente o método FindById. Podemos fazer isso porque TContaFactory retorna uma objeto TContaDao então podemos ter estes comandos aninhados.

Já no FindAll a única diferença é na hora da conversão devemos chamar ListToJSON, para convertermos toda a Lista de Objetos.

Server Container

Como último passo antes de passarmos para a aplicação cliente, vamos voltar ao ServerContainer e adicionar o componente TDSServerClass (**Figura 5**), apontar a propriedade Server para o nosso TDSServer e codificar a **Listagem 19** no evento OnGetClass.

Listagem 19. Implementação do evento OnGetClass

```
01 procedure TServerContainer.DSServerClassContaGetClass(
02   DSServerClass: TDSServerClass; var PersistentClass: TPersistentClass);
03 begin
04   PersistentClass := TSMConta;
05 end;
```

Feito isto basta rodar o servidor e deixar aberto para que possamos desenvolver a aplicação cliente.

Aplicação cliente

A aplicação cliente é relativamente simples, é uma interface desenvolvida com o Firemonkey que ficará como mostra a **Figura 6**. Para separarmos a parte dos dados podemos criar um DataModule que irá conter um componente SQLConnection que irá conectar no driver DataSnap.

Para gerarmos a classe Proxy basta clicarmos de direita no componente SQLConnection e escolher "Generated DataSnap Client Classes".

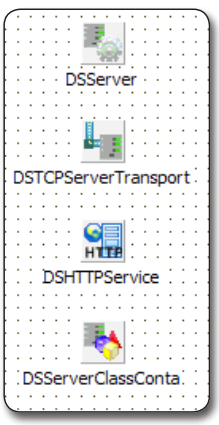


Figura 5. Componentes do ServerContainer

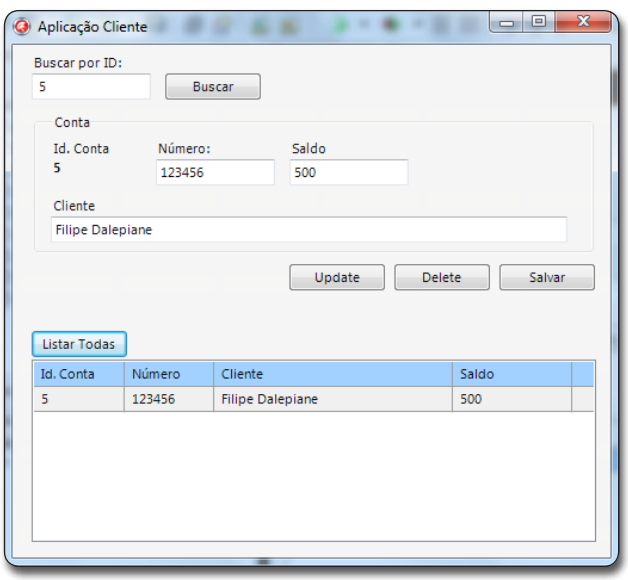


Figura 6. Interface da aplicação cliente

No evento OnCreate do formulário vamos criar uma instância da classe TSMContaClient, que foi gerada por nossa classe Proxy e configurar o Grid (**Listagem 20**).

Usaremos neste exemplo o componente TStringGrid para apresentar a lista de contas. Veja o código do evento OnClick do botão Listar Todos (**Listagem 21**). Temos uma variável i que é auxiliar apenas para incremento do número de linhas do Grid, após isso chamamos o método FindAll que buscará os dados do servidor, mas também usamos a classe TJSONUtil para convertermos o retorno que é em JSON para uma lista de Contas. Logo após utilizamos um ForIn para percorrermos os itens desta lista e adicionamos os dados neste grid.

Listagem 20. Evento OnCreate do Formulário

```
01 procedure TfrmMain.FormCreate(Sender: TObject);
02 begin
03  Server := TSMContaClient.Create(DM.DS_BANCO.DBXConnection);
04  with StringGrid1 do
05  begin
06   Cells[0, 0] := 'Id. Conta';
07   Cells[1, 0] := 'Número';
08   Cells[2, 0] := 'Cliente';
09   Cells[3, 0] := 'Saldo';
10   Columns[0].Width := 75;
11   Columns[1].Width := 75;
12   Columns[2].Width := 200;
13   Columns[3].Width := 100;
14  end;
15 end;
```

Listagem 21. Evento OnClick do botão Listar Todos

```
01 procedure TfrmMain.btnListarClick(Sender: TObject);
02 var
03  Conta: TConta;
04  i: Integer;
05 begin
06  i := 1;
07  ListaContas := TJsonUtil.JSONArrayToList<TConta>(Server.FindAll);
08  for Conta in ListaContas do
09  begin
10   StringGrid1.Cells[0, i] := IntToStr(Conta.Id);
11   StringGrid1.Cells[1, i] := Conta.Numero;
12   StringGrid1.Cells[2, i] := Conta.Cliente;
13   StringGrid1.Cells[3, i] := FloatToStr(Conta.Saldo);
14   Inc(i);
15   StringGrid1.RowCount := i;
16  end;
17  Conta.Free;
18 end;
```

O evento OnClick do botão buscar (**Listagem 22**) chama o método FindById da nossa classe Proxy.

Na **Listagem 23** temos os demais eventos para a manipulação. Os botões Update e Delete simplesmente chamam os métodos Update e Delete da classe Proxy, já o botão salvar cria uma nova instância de TConta antes de passar o objeto para a classe Proxy chamar o servidor para fazer a persistência.

Listagem 22. Evento OnClick do botão Buscar

```
01 procedure TfrmMain.btnBuscarClick(Sender: TObject);
02 begin
03  Conta := TJsonUtil.JSONToObject<TConta>
    (Server.FindById(StrToInt(NumberBox1.Text)));
04  if Conta = nil then
05   raise Exception.Create('Conta não encontrada');
06  lblIdConta.Text := IntToStr(Conta.Id);
07  edtNumero.Text := Conta.Numero;
08  edtCliente.Text := Conta.Cliente;
09  edtSaldo.Text := FloatToStr(Conta.Saldo);
10 end;
```

Listagem 23. Eventos dos Botões Delete, Update e Save

```
procedure TfrmMain.btnDeleteClick(Sender: TObject);
begin
 Server.Delete(TJsonUtil.ObjectToJSON<TConta>(Conta));
end;
procedure TfrmMain.btnUpdateClick(Sender: TObject);
begin
 Server.Update(TJsonUtil.ObjectToJSON<TConta>(Conta));
end;
procedure TfrmMain.btnSalvarClick(Sender: TObject);
begin
 Conta := TConta.Create;
 Conta.Numero := edtNumero.Text;
 Conta.Cliente := edtCliente.Text;
 Conta.Saldo := StrToFloat(edtSaldo.Text);
 Server.Save(TJsonUtil.ObjectToJSON<TConta>(Conta));
 btnListarClick(Self);
end;
```

Conclusão

O Delphi é uma linguagem atual e permite o uso de boas práticas. Como vimos no projeto, utilizamos o padrão DAO para isolar o acesso a dados, aplicamos injeção de dependência para criar uma aplicação desacoplada e tudo isso, numa aplicação feita com dbExpress e multicamadas. Além que, os dados entre servidor e cliente são transmitidos utilizando JSON, obtendo alto desempenho. Isso prova que não há desculpas para se desenvolver com qualidade. Não é porque sua aplicação seja totalmente orientada a objetos que você não pode fazer uso de suas boas práticas. O importante é entender os conceitos e adaptá-los à sua realidade.

Filipe Dalepiane
filipe.dalepiane@gmail.com
http://filipedalepiane.blogspot.com.br/
Analista Desenvolvedor Delphi a 5 anos, utilizando arquitetura multicamadas, com os Banco de Dados Firebird e DB2. Colunista da revista ClubeDelphi. Adepto de boas práticas de programação, modelagem UML, Orientação a Objetos e Design Patterns. Graduando Ciência da Computação pela UNIJUÍ.

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

RIA com Delphi – Parte 2

Aprenda criar aplicações ricas sem conhecer HTML

Este artigo faz parte de um curso

Resumo DevMan

RIA com Delphi:

É evidente a importância da Internet na atualidade. Diante deste fator, desenvolver aplicações para este ambiente é, sem dúvidas, um diferencial significativo. Entretanto, a Web conta com uma diversidade de recursos e, evidentemente, exige mais e mais conhecimento por parte da equipe de projeto. Contudo, existem diversos Frameworks para o próprio Delphi que visam facilitar e automatizar este processo, como é o caso do uniGUI e do Raudus. O uniGUI, conforme apresentado no primeiro artigo, permite que a aplicação RIA seja elaborada como se fosse destinada ao ambiente Desktop, sem que necessariamente o desenvolvedor conheça a linguagem HTML. Contudo, apesar disto, o uniGUI também permite que sejam adicionadas APIs, Scripts e recursos externos a aplicação, melhorando ainda mais a interatividade e, consequentemente, ampliando as funcionalidades da mesma. Este segundo artigo tem o intuito de demonstrar os passos necessários para aliar o uniGUI a outras tecnologias, inserindo códigos JavaScript e trabalhando com os recursos em conjunto, adotando como exemplo a Google Maps API, que permite aos desenvolvedores implementarem em suas aplicações recursos como a exibição de mapas e etc.

Em que situação o tema é útil:

Permitir que as aplicações para a Internet e Intranet sejam desenvolvidas rapidamente e de forma extremamente simples sem a necessidade de muitos conhecimentos do ambiente Web, utilizando os recursos do RAD e do Framework uniGUI, apresentado anteriormente, bem como estender as funcionalidades deste Framework adicionando APIs ou Scripts externos e empregando códigos JavaScript e HTML no projeto para aumentar ainda mais suas funcionalidades.

Na primeira parte deste artigo foram apresentadas algumas tecnologias e linguagens de programação que são empregadas no desenvolvimento de aplicações para a Internet, bem como o Framework ExtJS e uma simples visão do componente RIA para Delphi, uniGUI, demonstrando que o desenvolvimento destas aplicações é possível até mesmo sem domínio das linguagens básicas desta plataforma, o HTML e o JavaScript. Vale à pena relembrar que o uniGUI é baseado na biblioteca mencionada anteriormente, ExtJS, e conta com uma variedade de componentes visuais e não-visuais próprios com propriedades e eventos mais específicos para a elaboração de aplicações RIA, possibilitando a geração de Softwares autônomos, módulos ISAPI e serviços de aplicação para o Windows, além de ser gratuito, fácil de ser adicionado ao Delphi e gerar automaticamente toda a interface visual e recursos da aplicação para serem consumidos através de vários navegadores Web (Cross-Browser). Entretanto, o uniGUI não limita o desenvolvedor a utilizar apenas seus recursos e realizar todos os processos de forma automática. Através das funcionalidades características de seu conjunto de componentes, o desenvolvedor que desejar, pode facilmente adicionar recursos externos em sua aplicação, ou ainda criar manualmente Scripts, folhas de estilo em cascata (CSS), elementos HTML, possibilitando assim, a elaboração de aplicações ainda mais interativas e completas, unindo as linguagens Delphi, JavaScript e HTML para atender todas as necessidades do projeto. Sendo assim, mais uma vez o Framework uniGUI demonstra que é uma excelente ferramenta quando o assunto é o desenvolvimento RIA através do Delphi.

Outro ponto importante quando o assunto é integrar diversas linguagens em uma aplicação RIA é o emprego da tecnologia AJAX (Asynchronous JavaScript and XML). Este recurso é muito aplicado atualmente para a elaboração de WebApps e sites mais otimizados, uma vez que permite o emprego de requisições assíncronas entre o cliente e o servidor, possibilitando que um documento HTML possa ter seu conteúdo atualizado sem a necessidade de ser totalmente recarregado, dentre outras vantagens. Diante deste ponto de vista, podem ser criados eventos na aplicação RIA que rodem no lado cliente, através da linguagem JavaScript, por exemplo, que podem efetuar chamadas aos métodos do servidor para processar determinadas tarefas, de acordo com a necessidade do projeto.

Adição de recursos ao uniGUI

Como apresentado na primeira parte deste artigo, um dos objetos mais importantes do Framework uniGUI é o ServerModule,

que é criado automaticamente quando um novo projeto deste Framework é iniciado. Além da configuração básica do servidor, ele permite a definição da maior parte dos diretórios padrões que devem ser utilizados e possui outras diversas propriedades para que o desenvolvedor possa importar ou adicionar recursos à aplicação RIA. Dentre tais propriedades, algumas merecem destaque, sendo elas a CustomCSS, a CustomFiles e a CustomMeta. A CustomCSS permite que o desenvolvedor adicione à aplicação, folha de estilos em cascata (CSS) para formatar seus elementos ou adicionar outras funcionalidades possíveis com o emprego desta tecnologia. A CustomFiles, possibilita a adição de Scripts externos, tais como outras APIs JavaScript ou códigos manuais, permitindo ao desenvolvedor empregar Frameworks em conjunto. Já a CustomMeta tem o objetivo de permitir a adição de tags do tipo meta ao projeto, lembrando que este tipo de tag é utilizado para descrever o conteúdo de um site ou aplicação Web, algumas características entre outros. Sendo assim, o ServerModule é um dos principais meios para a adição de recursos à aplicação RIA.

Tutorial

uniGUI e Google Maps

O Google Maps pode ser incorporado nas aplicações de uma forma prática e rápida, onde as mesmas podem localizar as cidades utilizando as coordenadas geográficas, latitude e longitude, até mesmo sem que exista a necessidade da adição de componentes de terceiros, utilizando o objeto TWebBrowser, por exemplo, que permite a implementação de um navegador Web embutido. Entretanto, através do uniGUI, também é possível empregar o Google Maps na aplicação RIA de uma forma rápida. Para dar início a um simples exemplo prático, um novo projeto uniGUI deve ser iniciado (File>New>Other>uniGUI for Delphi>Application Wizard), mantendo a opção VCL application / Standalone Server selecionada, para a criação de uma aplicação autônoma. Após a criação do projeto, no formulário principal, MainForm, basta inserir um objeto do tipo TUniHTMLFrame, cujo objetivo é embutir na aplicação um quadro para a exibição de um conteúdo HTML, onde neste exemplo será utilizado para a visualização do mapa. Em seguida, devem ser adicionados quatro objetos do tipo TUniGroups para a organização dos controles. Um deles recebeu oito TUniSpeedButtons, um TUniTrackBar e um TUniSpinEdit, para interagir com os recursos do Google Maps, o segundo, um TUniListBox para a apresentação de uma lista de algumas cidades disponíveis, o terceiro, um TUniEdit e um TUniSpeedButton para possibilitar a pesquisa de novas cidades, e o quarto, dois objetos do tipo TUniLabel para apresentar as coordenadas de latitude e longitude.

Para utilizar o Google Maps na aplicação RIA é necessário adicionar manualmente sua API, baseada em JavaScript, para que a mesma seja carregada durante a inicialização. Para inserir Scripts adicionais ao projeto, conforme mencionado anteriormente, a propriedade CustomFiles, presente no objeto ServerModule, pode ser utilizada. Sendo assim, tal propriedade recebeu o endereço do script a seguir:

```
http://maps.googleapis.com/maps/api/js?v=3.9&sensor=false&language=pt-BR
```

Ao inserir este endereço, no momento em que a aplicação é inicializada, a API é obtida para que o Google Maps possa ser empregado na aplicação e, como pode ser observado neste trecho, alguns parâmetros também são informados, tais como a versão desejada da API (v=3.9) e a linguagem padrão (language=PT-BR).

A próxima etapa no projeto é inicializar o mapa e especificar onde o mesmo será carregado e exibido. Para tanto, no formulário principal (MainForm), o objeto UniHTMLFrame1, adicionado anteriormente, deve ser selecionado. A primeira etapa é atribuir em tal objeto, um elemento div identificado para que o mapa seja devidamente carregado nele através de JavaScript. O elemento div, presente na linguagem HTML e definido através das tags " ", é utilizado para criar uma seção ou divisão no documento HTML, em outras palavras, é um recipiente que pode conter outros elementos, de forma semelhante ao TPanel e TGroup do Delphi. Sendo assim, a propriedade HTML do objeto UniHTMLFrame1, recebeu o trecho de código a seguir:

```
<div id="area_mapa" style="height:100%; width:100%;"></div>
```

Neste simples trecho, o elemento de divisão (div) foi definido. Também foi informado um identificador para o mesmo através do atributo id, area_mapa, bem como definidas a largura e a altura com valores de 100% através da utilização do atributo style.

Com o elemento div criado para receber o mapa, o próximo passo é definir o código em JavaScript que será responsável por inicializar o mapa e apresentá-lo em tal elemento. Nos componentes do uniGUI é possível notar que existe uma variedade de propriedades e eventos incomuns aos tradicionais componentes do Delphi, onde os mesmos são utilizados para a adição de funcionalidades mais específicas para a Web, tal como a definição de Scripts por exemplo. Uma propriedade importante, presente em grande parte dos componentes do uniGUI, é a ClientEvents. Composta por outras subpropriedades. Através dela é possível definir diversos eventos que serão executados na máquina cliente, lembrando que a linguagem JavaScript é Client-Side. A **Figura 1** apresenta a janela para definição de eventos com scripts através da subpropriedade ExtEvents.

Como é possível notar, o objeto selecionado foi o UniHTMLFrame1 e existe uma diversidade de eventos que podem ser definidos através da mesma para a adição de funcionalidades na aplicação RIA. Sendo assim, para a devida inicialização do mapa, o evento Onafterupdatehtml, do objeto UniHTMLFrame1, recebeu um trecho de código em JavaScript. A **Listagem 1** apresenta o código que foi utilizado em tal evento.

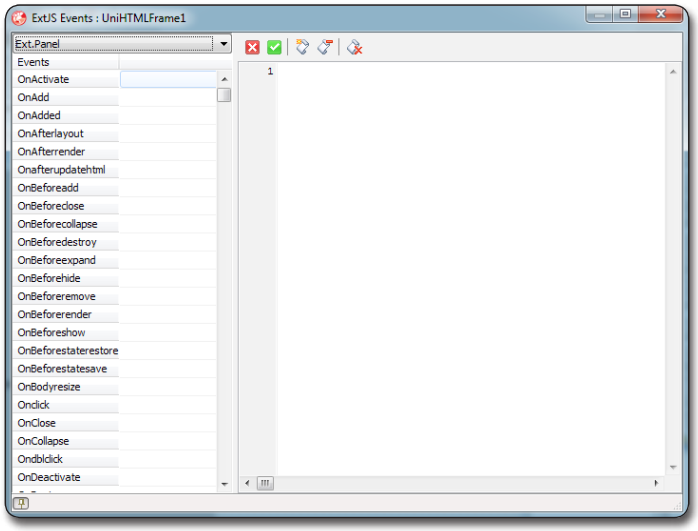


Figura 1. Janela ExtJS Events para a definição de eventos

Listagem 1. Código do evento Onafterupdatehtml do objeto UniHTMLFrame1

```
01 function Onafterupdatehtml(sender) {
02   var Coordenadas = new google.maps.LatLng(0.0, 0.0);
03   var mapa = new google.maps.Map(
04     document.getElementById("area_mapa"),
05     {
06       zoom: 1,
07       center: Coordenadas,
08       mapTypeId: google.maps.MapTypeId.ROADMAP
09     }
10   );
11   MapaGoogle = mapa;
12   google.maps.event.addListener(mapa, 'zoom_changed',
13   function() {
14     MainForm.UniTrackBar1.setValue(this.getZoom());
15   });
16   google.maps.event.addListener(mapa, 'click',
17    function(e) {
18     ajaxRequest(MainForm.UniHTMLFrame1,
19             'ClickMapa',
20             ['latitude='+e.latLng.lat(),
21             'longitude='+e.latLng.lng()]);
22    });
23 }
```

Inicialmente é declarada uma variável para armazenar as coordenadas (Latitude e Longitude – linha 02). Em seguida, é definida a variável "mapa" do tipo google.maps.Map, a classe da API do Google Maps responsável pela criação de um mapa em um elemento HTML (linha 03 a 10). Ela conta com um construtor que recebe como parâmetros, o local onde o mapa deve ser exibido, que normalmente é um elemento div, e as opções para o mapa, que permitem a definição do zoom, coordenadas iniciais, etc. Também é criada uma variável com a denominação de MapaGoogle para receber o mapa e facilitar os futuros acessos ao mesmo (linha 11). Posteriormente, é possível notar a adição de um Listener para o evento zoom_changed (alteração do zoom) do objeto mapa (linhas 12 a 15). O Listener é um objeto que implementa determinada interface para o tratamento de eventos, tais como manipulação de cliques sobre objetos, teclas pressionadas, etc. Este é um recurso amplamente empregado na linguagem de programação Java. No evento zoom_changed é possível observar que ao alterar o zoom do mapa, o valor do objeto UniTrackBar1 também é alterado de acordo com o valor atual do zoom, obtido através da função getZoom (linha 14). Para uma informação mais precisa, a propriedade Max do objeto UniTrackBar1 deve ter seu valor definido como 21, atualmente o valor máximo do zoom empregado para o mapa. Em seguida, é adicionado um novo Listener (linha 16) para o evento click do objeto mapa, cujo código atribuído efetua uma requisição AJAX chamando um método denominado "ClickMapa" e envia dois parâmetros para o mesmo, a latitude e a longitude. Sendo assim, o próximo passo é criar tal método (ClickMapa) para o objeto UniHTMLFrame1 (linhas 17 a 22), cujo intuito é simplesmente apresentar as coordenadas do mapa de acordo com o local onde o usuário clicou. O mesmo deve ser inserido no evento OnAjaxEvent de tal objeto, através da aba Events do Object Inspector. Este evento é muito interessante, pois, permite a integração de eventos entre cliente e métodos definidos no servidor. A **Listagem 2** apresenta o código empregado no evento OnAjaxEvent do objeto UniHTMLFrame1.

Listagem 2. Código do evento OnAnaxEvent do objeto UniHTMLFrame1

```
01 procedure TMainForm.UniHTMLFrame1AjaxEvent(Sender: TComponent;
     EventName: string; Params: TStrings);
02 begin
03   if EventName='ClickMapa' then
04   begin
05     UniLabel1.Caption := 'Latitude: '+Params.Values['latitude'];
06     UniLabel2.Caption := 'Longitude: '+Params.Values['longitude'];
07   end;
08 end;
```

Neste simples trecho de código é possível observar que variável parâmetro EventName é testada com o intuito de verificar se seu valor é equivalente a "ClickMapa" (linha 03), o nome atribuído para o método em questão, uma vez que poderiam ser criados outros neste mesmo evento. Satisfeita a condição, então a propriedade Caption do objeto UniLabel1 recebe o valor da latitude (linha 05), informado através da lista de strings, Params, e o mesmo processo ocorre com o objeto UniLabel2 (linha 06), porém para a exibição da longitude. Vale a pena lembrar que este evento é chamado através do Listener adicionado anteriormente em JavaScript para o objeto mapa.

Concluída esta etapa, o próximo passo é criar um novo método no projeto para possibilitar a alteração das posições atuais (latitude e longitude). A **Listagem 3** apresenta o método DefinirCoordenadas, criando no formulário MainForm, para definir a posição do mapa.

É possível observar a definição de três parâmetros para que as coordenadas e o zoom possam ser informados (linha 01). Em seguida, é adicionado um trecho de código JavaScript para a sessão atual através do método AddJS (linha 04). Dentro deste método,

primeiramente é verificado se a variável MapaGoogle é um objeto, e caso a condição seja satisfeita, então o método setCenter (linha 05) é chamado, recebendo as novas coordenadas (new Google.maps .LatLng(<latitude>,<longitude>);. O método setCenter de um objeto do tipo Map, presente na API do Google Maps, é utilizado para definir a posição atual do mapa, através da latitude e longitude. Também foi empregada a função StringReplace (linhas 06 a linha 09), para a substituição do caractere "," por "." em todas suas ocorrências (rfReplaceAll), bem como a função Format, utilizada para a formatação da latitude e longitude. Para concluir, também foi utilizado o método setZoom (linha 10), para definir o zoom atual do mapa. Criado o método para a definição das coordenadas, basta adicionar no momento da criação do formulário (OnCreate), uma chamada ao mesmo contendo alguns valores iniciais conforme o trecho a seguir:

```
DefinirCoordenadas(1, 1, 15);
```

Listagem 3. Código do método DefinirCoordenadas

```
01 procedure TMainForm.DefinirCoordenadas(
     Latitude, Longitude: Real; Zoom:  Integer);
02 begin
03   if WebMode then
04     UniSession.AddJS('if (typeof MapaGoogle=="object")
05   {MapaGoogle.setCenter(new google.maps.LatLng('+
06     StringReplace(Format('%2.4f', [Latitude]),",",
07     ".", [rfReplaceAll])+","+
08     StringReplace(Format('%2.4f', [Longitude]),",", 09  ".",
         [rfReplaceAll])+
10   ')); MapaGoogle.setZoom('+ IntToStr(Zoom) +');}');
11 end;
```

Neste simples trecho, foram definidas a latitude e a longitude com valor equivalente a um e o zoom igual a quinze.

A próxima etapa é atribuir os devidos eventos em JavaScript para os botões de navegação e definição do mapa. Para os botões de navegação, o método panBy pode ser utilizado. Este método permite efetuar uma alteração no centro do mapa através de valores x e y em pixels, ocorrendo uma movimentação. Como os eventos serão definidos em JavaScript, a subpropriedade ClientEvents.ExtEvents deve ser utilizada, assim como o evento OnClick, presente na janela ExtJS Events. Sendo assim, no primeiro botão de navegação, btNorte, para realizar o movimento no sentido Norte, basta inserir o trecho de código MapaGoogle.panBy(0, -25), no segundo, sentido Leste, o código MapaGoogle.panBy(25, 0), no terceiro, sentido Sul, MapaGoogle.panBy(0, 25), e o último, sentido Oeste, o código MapaGoogle.panBy(-25, 0). Definidos os botões de navegação, também devem ser definidos os botões para as diferentes exibições do mapa, também utilizando a janela ExtJS Events. Sendo assim, o evento OnClick do primeiro botão, denominado btHibrido, que apresenta tanto o mapa com imagens de satélite quanto a malha viária, deve receber o trecho de código apresentado a seguir:

```
MapaGoogle.setOptions({mapTypeId: google.maps.MapTypeId.HYBRID});
```

Como pode ser observado, o método setOptions foi empregado. De fato, tal método permite alterar as opções para o mapa em questão, e de acordo com o trecho apresentado, o id do tipo de mapa (mapTypeId) é alterado para o valor do id referente ao mapa Híbrido, através da constante google.maps.MapTypyId.HYBRID. Sendo assim, os três botões restantes devem receber o mesmo trecho de código, substituindo apenas a constante HYBRID pela outra constante que representa o Id do mapa em questão, conforme a estrutura google.maps.Map.<Constante>. A **Tabela 1** apresenta as constantes que podem ser empregadas para alterar o tipo do mapa.

Constante	Descrição
HYBRID	Exibição do mapa em modo híbrido (satélite e malha viária).
ROADMAP	Exibição do mapa rodoviário com informações.
SATELLITE	Exibição do mapa baseado em imagens de satélite.
TERRAIN	Exibição do mapa em relevo físico.

Tabela 1. Constantes para a exibição de um mapa

Adicionados os tipos de mapa para cada botão, o próximo passo é inserir outros eventos baseados em JavaScript para os objetos UniTrackBar1 e UniSpinEdit1, permitindo uma interação dos mesmos com o mapa, também através da janela ExtJS Events. Sendo assim, o evento OnChange do objeto UniTrackBar1 deve receber o trecho de código apresentado a seguir:

```
MapaGoogle.setZoom(newValue);
MainForm.UniSpinEdit1.setValue(newValue);
```

Neste trecho, primeiramente é definido o zoom para o mapa através do método setZoom, recebendo como parâmetro, o novo valor definido através da barra, lembrando que o valor máximo configurado foi 21 através da propriedade Max. Em seguida, também é atualizado o valor do objeto UniSpinEdit1 através do método setValue para que ambos estejam sempre com o mesmo valor. Sendo assim, o objeto UniSpinEdit1 também deve receber um simples trecho de código para interagir com o UniTrackBar1. Seu evento OnSpin recebeu o seguinte código:

```
MainForm.UniTrackBar1.setValue(sender.field.getValue());
```

É possível observar que, ao alterar o valor do UniSpinEdit1, o mesmo também altera o valor do objeto UniTrackBar1, e consequentemente, o zoom do mapa.

Concluída esta etapa, o próximo passo é criar uma simples lista de cidades para o objeto UniListBox1 para que, conforme o usuário clique sobre a mesma, a cidade selecionada seja carregada no mapa. De fato, como o Google Maps é baseado em coordenadas, para carregar a devida cidade selecionada pelo usuário, basta apenas informar a latitude e longitude da mesma e utilizar o método setCenter, onde neste exemplo basta chamar o procedimento criado anteriormente, DefinirCoordenadas.

Contudo, antes de definir o código para o evento, a propriedade Items do objeto UniListBox1 foi definida com os valores Monte Alto, São Paulo e Rio de Janeiro. Em seguida, foi atribuído o código para o evento. A **Listagem 4** demonstra o código empregado para o evento OnClick do objeto UniListBox1 para o carregamento da cidade selecionada.

Listagem 4. Evento OnClick do objeto UniListBox1

```
01 procedure TMainForm.UniListBox1Click(Sender: TObject);
02 begin
03   case UniListBox1.ItemIndex of
04     0: DefinirCoordenadas(-21.26254812987739, -48.49859923782043, 13);
05     1: DefinirCoordenadas(-23.54914000643578, -46.62941973961182, 10);
06     2: DefinirCoordenadas(-22.900455772288616, -43.208642762426734, 10);
07   end;
08 end;
```

Como pode ser observado, este simples trecho de código verifica qual é o item selecionado através da propriedade ItemIndex (linha 03), e de acordo com o valor do mesmo, chama o método DefinirCoordenadas (linhas 04 a 06) para informar a latitude, a longitude e o zoom para a cidade em questão.

O próximo passo é elaborar o código para efetuar a pesquisa de uma rua ou cidade através de seu nome, estado e país. Para tanto, a API do Google Maps conta com uma classe denominada Geocoder, podendo efetuar um processo denominado Geocodificação. Este recurso possibilita a conversão de um endereço ou do nome da cidade/local para suas devidas coordenadas, retornando o resultado em forma de um objeto literal, escrito na notação JSON (JavaScript Object Notation). Sendo assim, para possibilitar a pesquisa, foi adicionado um código ao botão de busca inserido anteriormente, btPesquisar. A **Listagem 5** apresenta o código para efetuar a busca de um endereço ou cidade.

Ao efetuar um clique neste botão, é adicionado um código em JavaScript na sessão atual, ocasionando sua execução. Neste código, inicialmente é criado um novo objeto do tipo Geocoder (linha 03). Posteriormente, o método geocode do mesmo é chamado para que a pesquisa seja efetuada (linha 04), onde o parâmetro address (endereço) é preenchido de acordo com o texto informado no objeto edPesquisa, e após a execução do processo, as variáveis resultado e status recebem os dados de retorno (linhas 05 e 06). Após a conclusão do processo, a variável status é testada de acordo com a constante google.maps.GeocoderStatus.OK para verificar se a cidade ou rua pesquisada foi encontrada. Caso tenha sido, então

CURSOS ONLINE

A Revista Clube Delphi oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Clube Delphi

CONHEÇA ALGUNS DOS CURSOS:

- Delphi para Iniciantes
- Criando componente Boleto em Delphi
- Loja Virtual em Delphi Prism, Rave Reports – Impressão de Gráficos Personlizados
- Aplicação para repasse de Dados com Delphi

Para mais informações :
www.devmedia.com.br/cursos/delphi
(21) 3382-5038

DevMedia

o método setCenter (linha 07) é chamado e, através do resultado retornado, são definidas as novas coordenadas para a exibição do mapa de acordo com o local localizado. Também é definido um zoom para o mapa através do método setZoom (linha 08). Para concluir, se a cidade não foi encontrada, então é apresentada uma mensagem para o usuário (alert) (linha 10), contendo as informações de retorno presentes na variável status.

Para finalizar a aplicação, outro simples processo que pode ser realizado para personalizar o ambiente RIA é definir um tema para o mesmo. Tal processo pode ser realizado através da propriedade Theme, presente no objeto MainModule. Para este exemplo, tal propriedade foi definida com o valor "Access". A **Figura 2** apresenta a aplicação RIA em execução, acessada diretamente pelo navegador.

Listagem 5. Evento OnClick do botão btPesquisar

```
01 procedure TMainForm.btPesquisarClick(Sender: TObject);
02 begin
03  UniSession.AddJS('geocoder = new google.maps.Geocoder();'+
04  'geocoder.geocode( {"address":"'+ edPesquisa.Text +'"},
05  function(resultado, status) {'+
06'  if (status == google.maps.GeocoderStatus.OK) {'+
07'   MapaGoogle.setCenter(resultado[0].geometry.location);'+
08'   MapaGoogle.setZoom(8);'+
09'  } else {'+
10'   alert("Erro: " + status);'+
11'  } });');
12 end;
```

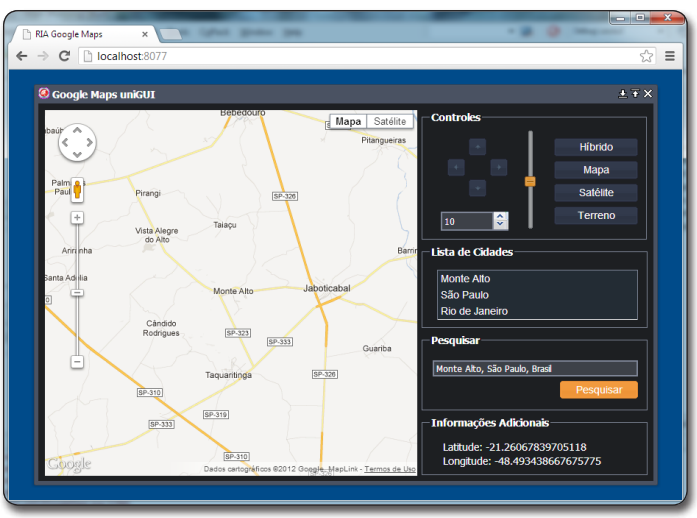


Figura 2. Aplicação uniGUI RIA integrada ao Google Maps em execução

Como pode ser observado, o uniGUI é um Framework muito flexível que possibilita que suas funcionalidades sejam facilmente estendidas para que o desenvolvedor possa utilizar os recursos que necessitar em seu projeto.

Conclusão

O desenvolvimento de aplicações para a Internet e Intranet é um processo muito simplificado e rápido através do excelente Framework uniGUI. Esta ferramenta é capaz de gerar automaticamente toda a codificação HTML e JavaScript para a aplicação RIA, não exigindo o domínio destas linguagens por parte da equipe, mas também oferece a possibilidade do desenvolvedor adicionar funcionalidades externas e manuais ao seu Software de acordo com suas necessidades. Como foi apresentado neste artigo, com a definição de algumas propriedades e eventos, bem como a elaboração de métodos, a aplicação RIA uniGUI foi facilmente integrada a API do Google Maps. Sendo assim, para a adição de quaisquer recursos externos aos futuros projetos com este Framework, uma alternativa é seguir os mesmos passos que foram apresentados.

Notavelmente, o uniGUI além de prover toda a base necessária para a criação de uma aplicação RIA através do Delphi, ainda permite que funcionalidades sejam adicionadas, ampliando ainda mais as possibilidades do Software, confirmando sua flexibilidade e utilidade para projetos desta categoria.

Giuliano Scombatti Pinto
giuliano@sygnux.com.br
É Analista Desenvolvedor da Sygnux Software (www.sygnux.com.br) localizada em Monte Alto/SP, atuando também com desenvolvimento Web e Android. Formado pela Fatec/TQ, começou a desenvolver com Delphi em 2003. Atualmente trabalha com Delphi, Java, PHP e Flash/Flex. Professor de Informática de Projetos Sociais da Prefeitura Municipal de Monte Alto/SP.

Maiores informações e download do uniGUI
http://www.unigui.com/

Maiores informações e download do Raudus
http://www.raudus.com/

Maiores informações e download do ExtJS
http://www.sencha.com/products/extjs/

Maiores informações sobre a API JavaScript do Google Maps
https://developers.google.com/maps/documentation/javascript/reference

DevSpace do Autor
http://www.devmedia.com.br/space.asp?id=283274

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Generics + MVC = Boas práticas e reusabilidade

Indo além do conceitual

Ao longo deste artigo veremos como utilizar o recurso de generics, ou tipos genéricos, no Delphi. Os tipos genéricos foram introduzidos ao Delphi a partir de 2009. Com tipos genéricos nós podemos parametrizar determinados tipos, deixando os mesmos para serem definidos somente no momento de seu uso, isso nos traz certa flexibilidade, pois podemos criar classes mais abstratas para determinados tipos de objetos sem nos preocuparmos com qual tipo exato estaremos trabalhando além de reduzir consideravelmente a necessidade de type-casting.

Após passarmos pelos conceitos e exemplos isolados de Generics, criaremos um exemplo onde simularemos um cenário de um projeto MVC – Model View Controller – (ver **Nota do DevMan 1**) onde teremos um controller genérico que espera dois parâmetros, sendo um Form e o outro um objeto de nosso domínio. Para isso criaremos um pequeno cenário com dois cadastros básicos, um de cliente e um de fornecedor. Em nosso exemplo não chegaremos a persistir os dados, pois o objetivo do artigo é demonstrar o uso do generics e fazer o leitor perceber os benefícios do uso do mesmo.

Com este exemplo conseguiremos compreender o uso e os benefícios do Generics além de abordar o uso de constraints para nossos tipos genéricos.

Nota do DevMan 1

MVC, Model View Controller, é um padrão de projeto que prevê a separação da aplicação em três camadas lógicas, Model, View e Controller. A ideia é seguir o princípio da responsabilidade única e garantir que as classes da camada Model cuidem apenas dos aspectos relacionados ao modelo, ou seja, manutenção do estado, regras de negócio, comportamento, abstrações do mundo real etc... Enquanto que a camada View cuida apenas da apresentação para o usuário final, sem conhecer detalhes do sistema e sem possuir regras, a camada View deve possuir apenas elementos necessários para comunicação com o usuário final. E por fim a camada controller seria a responsável pela comunicação entre a View e o Model. Este padrão é bastante conhecido e difundido e tem como um dos principais benefícios a separação de responsabilidades e com isso um maior nível de desacoplamento e reuso.

Resumo DevMan

Generics + MVC = Boas práticas e reusabilidade:

Neste artigo criaremos um aplicativo Win32 que utiliza o padrão MVC para mostrar como o recurso de Generics no Delphi pode trazer vantagens. Vamos além dos exemplos simples de listas tipadas, mostrando como o recursos pode ser útil no dia a dia. Claro, explicaremos o que são esses tipos e como eram as coisas antes do seu surgimento, abordando os benefícios de seu uso em classes e interfaces, inclusive com uso de constraints. Veremos também como o próprio Delphi faz uso deste recurso em diversos elementos de sua arquitetura. Além disso, discutiremos um pouco sobre covariância e contravariância para explicitar limitações que o uso de generics ainda possui no Delphi.

Em que situação o tema é útil:

Generics podem ser utilizados em quaisquer projetos que venhamos a desenvolver. Através deles podemos tornar nossas arquiteturas mais abstratas e reutilizáveis além de reduzir a quantidade de código de nossos projetos através da eliminação de códigos redundantes.

O que é Generics ?

O recurso de tipos genéricos permite a definição tardia de determinados tipos de objetos. Sendo assim, nós podemos parametrizar nossas classes, interfaces e métodos informando para elas com qual tipo irão trabalhar somente no momento do uso das mesmas.

Podemos retirar um belo exemplo (**Listagem 1**) disso utilizando um tipo do próprio Delphi, o TList da unit System.Generics.Collections. Antes do TList genérico no Delphi, precisávamos trabalhar com o TObjectList e normalmente necessitávamos realizar type casting (ver **Nota do DevMan 2**) para converter os objetos que eram recuperados do TObjectList, visto que ele trabalhava com instâncias de TObject. Agora com o TList genérico, no momento em que declaramos uma variável deste tipo informamos com qual tipo o TList irá trabalhar, dessa forma, todos os métodos de TList trabalharão com este tipo, inclusive os retornos dos métodos que usamos para recuperar elementos da lista, já retornarão o objeto no tipo correto.

No exemplo da **Listagem 1** declaramos um TList de strings e então os parâmetros dos métodos do nosso objeto MinhaLista só aceitarão String. Isto garante que nossa lista só trabalhará com objetos do tipo string e reduz a chance de obtermos um erro por armazenar um objeto de outro tipo na lista.

Outro benefício que podemos ver no exemplo acima é quando recuperamos um item da lista. Como já definimos o tipo da lista na declaração, não precisamos nos preocupar em fazer um Type Casting do item quando recuperamos o mesmo.

Listagem 1. Exemplo de lista genérica

```
procedure TFrmPrincipal.Teste;
  var MinhaLista:TList<String>;
     itemLista:String;
begin
  MinhaLista :=TList<String>.Create();
  MinhaLista.Add('Texto 1');
  MinhaLista.Add('Texto 2');
  MinhaLista.Add('Texto 3');
  MinhaLista.Add('Texto 4');
  itemLista := MinhaLista[0];
end;
```

Nota do DevMan 2

Type casting é como chamamos o recurso de conversão de um determinado tipo para outro. Por exemplo, se temos uma variável "obj" declarada como TObject porém que recebeu uma instância de TCliente, se tentarmos acessar as propriedades de cliente em "obj" não conseguiremos, para isso precisaremos fazer um typecast para o tipo TCliente. Vale ressaltar que só é possível realizar o typecasting de elementos compatíveis com a instância da declaração. Ou seja, se neste caso tentarmos converter "obj" em um TProduto, vamos obter uma exception de "invalid type casting".

Utilizando tipos genéricos

Os tipos genéricos podem ser atribuídos em Classes, Interfaces, Records e Métodos. A definição mais simples de um tipo genérico é feita simplesmente inserindo o nome de referência para o tipo entre "<>". Por exemplo, como mostra o código a seguir.

```
function TestGenerics<T>(): T;
var variavelTeste:T;
begin
  //FAZ ALGUMA COISA
  result := variavelTeste;
end;
```

Temos o exemplo simples de um método com uso de um tipo genérico, onde T é o nome de seu tipo genérico. Veja que o próprio tipo que é passado como parâmetro é usado como retorno do método e como definição para a variável variavelTeste. O retorno deste método irá variar de acordo com o tipo passado como parâmetro para ele. Para executar o método, basta passarmos o tipo como parâmetro, como mostrado a seguir:

```
TestGenerics<String>();
```

Para a definição na classe, teríamos a mesma sintaxe, como no exemplo da **Listagem 2**.

Poderíamos utilizar esta classe de duas formas:

• Declarando uma variável do tipo da classe - Neste caso, teríamos que declarar a variável especificando o tipo que a classe trabalharia. Por Exemplo:

```
procedure TestUsandoGenerics();
var variavelTeste: TTestGenerics<String>;
begin
  variavelTeste := TTestGenerics<String>.Create();
end
```

• Herdando em uma classe mais específica - Neste caso, no momento da herança, teríamos que especificar qual o tipo utilizado pela classe. Por exemplo:

```
TTestGenericsClasseFilha = class(TTestGenerics<String>)
  //Classe Filha
end;
```

Listagem 2. Definição de tipo genérico em uma classe

```
type
  TTestGenerics<T> = class(TObject)
  private
    FAtributo: T;
    procedure SetAtributo(const Value: T);
  published
    property Atributo:T read FAtributo write SetAtributo;
  end;

implementation

procedure TTestGenerics<T>.SetAtributo(const Value: T);
begin
  FAtributo := Value;
end;
```

Se não quisermos definir o tipo na definição da herança da classe filha e desejarmos postergar esta definição para o momento de uso da mesma, basta definirmos um tipo genérico para ela e repassar o tipo para a classe pai, como no exemplo a seguir:

```
TTestGenericsClasseFilha<X> = class(TTestGenerics<X>)
  //Classe Filha
end;
```

Nos exemplos mostrados até agora, qualquer tipo que for passado como parâmetro será aceito. Em alguns casos isso não é problema, porém em outros pode gerar certas dores de cabeça. O Delphi permite que sejam definidas **constraints** para os tipos genéricos. Estas constraints restringem os tipos possíveis de serem passados como parâmetros. Estas podem ser:

• Exigir que o tipo passado como parâmetro implemente uma ou várias interfaces.

• Exigir que o tipo passado como parâmetro herde de uma determinada classe

• Uma das seguintes palavras reservadas:
- class - Indica que o tipo passado como parâmetro precisa ser um tipo de referencia, ou seja, uma Classe ou uma Interface. Isso indica que não poderá ser passado um tipo primitivo (inteiro, double etc...) como parâmetro para este tipo genérico.
- constructor - Quando usada esta constraint, o Delphi exigirá que o tipo passado como parâmetro tenha um construtor público padrão (Create), ou seja, sem parâmetros.
- record - Indica que o tipo passado como parâmetro não pode ser um tipo de referencia, ou seja, precisa ser um Value Type, sendo um Record ou um tipo primitivo (inteiro, Double etc...).

Colocando em prática, criando a camada model

Para nosso exemplo foi criado um novo projeto VCL. Vamos primeiramente criar uma classe base para nossos modelos, para isso, adicionamos uma nova unit ao projeto, nomeando como UBaseModel e nela criamos a classe TBaseModel. Declare uma property Id a esta classe, conforme demonstrado na **Listagem 3**.

Listagem 3. Modelo base

```
unit UBaseModel;

interface

uses System.Generics.Collections;

type
 TBaseModel = class(TObject)
 private
  FId: Integer;
  procedure SetId(const Value: Integer);
 published
  property Id: Integer read FId write SetId;
 end;

implementation

{ TBaseModel }

procedure TBaseModel.SetId(const Value: Integer);
begin
 FId := Value;
end;

end.
```

A ideia é que todos os nossos modelos tenham um identificador artificial. Em todo domínio, existem atributos que são candidatos a chaves naturais para nossos modelos, por exemplo, CPF, CNPJ, Cod. Produto, etc. Porém é uma boa prática termos um identificador artificial, que servirá apenas para identificar o objeto e manter o relacionamento do mesmo com outras entidades no banco de dados. Este identificador uma vez atribuído a uma entidade, jamais será alterado, diferentemente de uma chave natural que poderá sofrer mudanças.

Agora, vamos criar duas classes de domínio, TCliente e TFornecedor, que herdam de TBaseModel. Elas possuem campos básicos, pois o objetivo do exemplo é garantir o mínimo de entendimento sobre Generics. Então adicione duas novas units ao projeto e nomeie as mesmas como UCliente e UFornecedor, respectivamente, adicione o código mostrado nas **Listagens 4** e 5.

Listagem 4. Classe TCliente

```
unit UCliente;

interface

uses UBaseModel;

type
 TCliente = class(TBaseModel)
 private
  FAtivo: Boolean;
  FIdade: Integer;
  FNome: String;
  procedure SetAtivo(const Value: Boolean);
  procedure SetIdade(const Value: Integer);
  procedure SetNome(const Value: String);
 published
  property Nome:String read FNome write SetNome;
  property Idade:Integer read FIdade write SetIdade;
  property Ativo:Boolean read FAtivo write SetAtivo;
 end;

implementation

//o código da implementação pode ser obtido no download do projeto

end.
```

Listagem 5. Classe TFornecedor

```
unit UFornecedor;

interface

uses UBaseModel;

type
 TFornecedor = class(TBaseModel)
 private
  FCNPJ: String;
  FRazaoSocial: String;
  FNomeFantasia: String;
  procedure SetCNPJ(const Value: String);
  procedure SetNomeFantasia(const Value: String);
  procedure SetRazaoSocial(const Value: String);
 published
  property NomeFantasia:String read FNomeFantasia write SetNomeFantasia;
  property RazaoSocial:String read FRazaoSocial write SetRazaoSocial;
  property CNPJ:String read FCNPJ write SetCNPJ;
 end;

implementation

//o código da implementação pode ser obtido no download do projeto

end.
```

Todas as classes de domínio de nosso projeto irão herdar de TBaseModel. Esta é uma maneira interessante de manter certos padrões, como por exemplo, o próprio Id do objeto. Além disso, facilitaria futuras implementações de auditoria (usuário/dia/hora de criação e alteração), pois poderíamos implementar isto apenas na classe base. Sem contar que possibilita restringirmos o uso de determinados métodos apenas às classes de domínio reduzindo as chances de serem passados objetos inesperados como parâmetros para as classes de nossa arquitetura.

Criando a camada view

Vamos criar agora as telas de nosso projeto. Teremos apenas quatro formulários para nossos exemplos.

- BaseForm - Formulário base para os cadastros que terá um botão de salvar e um contador de registros "salvos", visto que não teremos persistência em nosso exemplo.
- FrmPrincipal - Formulário principal do projeto que terá as opções para acesso aos cadastros de cliente e fornecedor.
- FrmCadastroCliente - Formulário de cadastro de clientes, que deverá herdar de BaseForm e será controlado pelo ClienteController.
- FrmCadastroFornecedor - Formulário de cadastro de fornecedores, que deverá herdar de BaseForm e será controlado pelo FornecedorController.

Primeiramente vamos criar nosso formulário base. Para isso, clique com o botão direito sobre o projeto, opção Add New VCL Form, nomeie a unit como UBaseForm. Feito isso, renomeie o formulário para FrmBaseForm e adicione um label e um botão no mesmo, de forma que fique com a aparência semelhante à **Figura 1**.

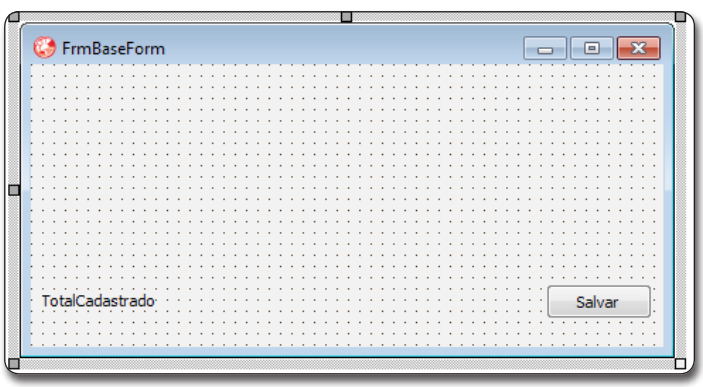


Figura 1. Formulário base

Nomeie o label como lblTotalCadastrado e o Button como btnSalvar. Isto basta para nosso formulário base.

Vamos criar agora nossos formulários de Cadastro de Cliente e Fornecedores. Esses formulários terão um processo de criação diferente do BaseForm, pois os mesmos serão herdados de BaseForm usando o recurso de herança visual do Delphi. O recurso de herança visual nos permite facilmente padronizar nossas interfaces com o usuário.

Para criar um formulário herdado, clique com o botão direito sobre o projeto, opção Add New - Other. Uma janela se abrirá, selecione a opção Inheritable Items, dentro de Delphi Projects na árvore do lado esquerdo e selecione FrmBaseForm na área que aparecerá no lado direito da tela.

Nomeie este formulário como FrmCadastroCliente e repita o procedimento para o cadastro de Fornecedores, nomeando o mesmo como FrmCadastroFornecedor.

Inclua controles nestes formulários de forma a atender os modelos de Cliente e Fornecedor que criamos anteriormente. Os formulários terão que ter aparência semelhante a estas das **Figuras 2** e **3**.

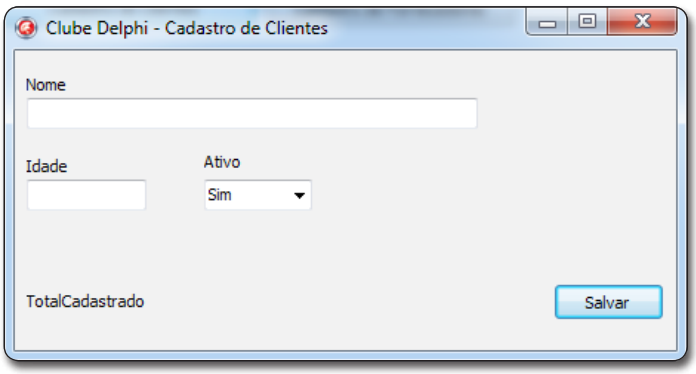


Figura 2. Cadastro de Clientes

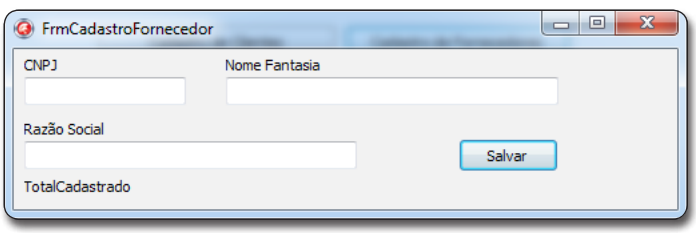


Figura 3. Cadastro de Fornecedores

Altere o nome dos controles de forma que fiquem assim:

- Cadastro de Clientes
  - edtNome
  - edtIdade
  - cbbAtivo

- Cadastro de Fornecedores
  - edtCNPJ
  - edtFantasia
  - edtRazaoSocial

Vale ressaltar a importância de utilizarmos as convenções para nomenclatura dos componentes, como visto, TEdit com prefixo edt, TComboBox com prefixo cbb, TButton com prefixo btn, dessa forma a manutenção fica facilitada e toda vez que estes componentes forem usados no código, o programador já saberá qual o tipo do mesmo.

Criando a camada controller

Agora que já criamos nossos modelos e nossas views, vamos implementar nossa camada controller, que é justamente onde aplicaremos o Generics. Temos um controller base, que receberá dois tipos genéricos, um para indicar o formulário e o outro para indicar o modelo nos quais ele terá que controlar.

Adicione uma nova unit ao projeto, chamada UBaseController. Defina a interface da classe de forma que ela fique como na **Listagem 6**.

Listagem 6. Interface da classe TBaseController

```
01 unit UBaseController;
02
03 interface
04
05 uses UGenericDAO, System.Generics.Collections, Vcl.Forms, UBaseForm,
    UBaseModel;
06
07 type
08  TBaseController<TFormulario: TFrmBaseForm; TModelo:TBaseModel > =
09 class(TObject)
10  private
11   FFormulario:TFormulario;
12   FModelo: TModelo;
13   FModelosCadastrados: TList<TModelo>;
14  public
15   procedure ShowFormModal(modelo: TModelo);
16   procedure ClearControls();virtual;abstract;
17   procedure BindInView();virtual;abstract;
18   procedure BindInModel();virtual;abstract;
19   procedure Save();
20   function GetTotalCadastrados():Integer;
21   constructor Create();
22   destructor Destroy();override;
23   procedure SalvarClick(sender : TObject);
24   procedure CloseForm(Sender: TObject; var Action: TCloseAction);
25  end;
```

Observe que na definição da classe TBaseController, indicamos quais os tipos parametrizados que teremos:

```
TBaseController<TFormulario:TFrmBaseForm;TModelo:TBaseModel>
```

Definimos a classe TBaseController esperando dois tipos genéricos, TFormulario e TModelo. Observe que ambos possuem uma constraint. O tipo genérico TFormulario possui uma constraint indicando que o mesmo deverá obrigatoriamente herdar de TFrmBaseForm enquanto que o tipo genérico deverá obrigatoriamente herdar de TModelo. Com isso garantimos que nossos controllers só serão usados com classes que herdam destes tipos, evitando erros em tempo de execução.

Temos também na seção Private (linha 10) três variáveis definidas:

• FFormulario - Esta variável armazenará a instância do formulário gerenciado pelo Controller.

• FModelo - Esta variável armazenará a instância do modelo gerenciado pelo controller.

• FModelosCadastrados - Lista com os modelos "cadsatrados". Lembrando que não teremos persistência em nosso exemplo.

Na seção Public (linha 14) temos os métodos para controle do formulário e do modelo. Observe os que são abstratos e virtuais (linha 16, 17 e 18). Estes métodos dependem do conhecimento dos modelos e das views concretas, com isto não podemos implementar os mesmos em nosso BaseController, deixando que esta implementação seja feita pelos controllers concretos. O objetivo destes métodos é o seguinte:

• ClearControls – Limpar os controles do formulário gerenciado pelo controller

• BindInView – Ler as propriedades do modelo e renderizar nos controles da view.

• BindInModel – Ler os controles da view e inserir os dados nas propriedades do modelo.

Na **Listagem 7** vemos a implementação do Create e do Destroy. No Create nós instanciamos nossa lista de modelos cadastrados e em seguida criamos o formulário, de acordo com o tipo do formulário de definido no tipo genérico. Após isso atribuímos o evento click do botão salvar ao nosso método SalvarClick e o onClose do formulário ao nosso método CloseForm.

Listagem 7. Implementação do construtor e destrutor do BaseController

```
constructor TBaseController<TFormulario, TModelo>.Create();
begin
 FModelosCadastrados := TList<TModelo>.Create();
 Application.CreateForm(TFormulario, FFormulario);
 FFormulario.BtnSalvar.OnClick := SalvarClick;
 FFormulario.OnClose := CloseForm;
end;

destructor TBaseController<TFormulario, TModelo>.Destroy;
begin
 FFormulario.Release();
 FModelo.Destroy();
 inherited;
end;
```

No destrutor nós liberamos os objetos criados da memória. Um detalhe importante a se destacar é o uso da palavra reservada "override" na declaração do destrutor. Este é um detalhe muito importante, pois o método Destroy da classe TObject (Base de todas as classes do Delphi), é virtual. Um erro muito comum dos desenvolvedores e que causa muita dor de cabeça, é a não definição do destroy como override. Isso faz com que, caso a variável de referência seja do tipo TObject, porém a instancia seja do seu tipo concreto (que possui um destroy sem override), ao chamar variável.destroy, será executado o método Destroy de TObject e não de sua classe concreta, ocasionando um memory leak (objeto instanciado mas não liberado) em seu projeto.

Na **Listagem 8** temos a implementação de outros métodos do BaseController, neste caso é importante destacarmos CloseForm, que seta a action como caHide, libera a referência do modelo e limpa os controles do formulário. Em seguida temos o método Save que "salvaria" o objeto no banco de dados.

Nota do DevMan 3

Design patterns são padrões de projeto que preveem a definição de soluções comuns para problemas recorrentes. Cada design pattern provê a solução de um problema específico. Os design patterns são independentes de tecnologia e podem ser implementados em qualquer linguagem que suporte orientação a objetos. Existem diversos padrões para os mais variados problemas, sendo que os mais famosos estão catalogados no livro conhecido como GOF (Gang of Four) e se dividem em três grandes grupos:

- **Patterns Estruturais** – São aqueles que cuidam do relacionamento entre as entidades do projeto, fazendo com que as mesmas tenham suas estruturas mais claras e desacopladas, visando a solução de um determinado problema. Ex.: Proxy, Facade, Adapter etc...

- **Patterns Criacionais** – São aqueles que cuidam de estratégias de criação de objetos, fornecendo uma linguagem única e um local centralizado para criação de determinados tipos de objetos. Ex.: Singleton, Builder, Abstract Factory etc...

- **Patterns Comportamentais** – São aqueles que cuidam da colaboração entre as entidades do seu projeto, deixando esta comunicação mais fluída e desacoplada entre as mesmas. Ex.: Strategy, Observer, Command etc....

Nota do DevMan 4

Template Method é um design pattern comportamental que prevê a definição de um esboço do algoritmo de determinada rotina, possibilitando que pedaços deste algoritmo sejam sobrescritos em classes filhas, garantindo que o algoritmo principal será executado. O template method parte do princípio de que a classe base conhece o que precisa ser feito, porém não tem conhecimento sobre o como fazer, delegando tal responsabilidade, em determinados passos do algoritmo, para suas classes filhas.

Em nosso exemplo ele simplesmente faz um bind do que foi digitado para o objeto, verificamos se é um objeto novo, através da verificação se o Id é igual a 0. Caso seja novo, incluímos em nossa lista de modelos cadastrados e atualizamos a label de total no formulário.

É interessante observar que mesmo sem ter conhecimento de qual modelo e com qual formulário iremos trabalhar, conseguimos implementar um algoritmo básico, deixando que as partes concretas sejam definidas pelos controllers concretos, isto caracteriza o uso do design pattern chamado template method (ver **Notas do DevMan 3 e 4**).

Agora que temos nosso BaseController implementado, precisamos criar nossos controllers concretos, com a definição dos tipo utilizados e com a implementação dos métodos abstratos. Para isso, adicione duas novas units e as nomeie como ClienteController e FornecedorController. As mesmas terão a implementação conforme as **Listagens 9** e **10**, respectivamente. Os controllers criados têm a responsabilidade de passar para os controles visuais os dados dos objetos e vice-versa.

Na **Listagem 9** temos o controller do cadastro de cliente, onde temos o método BindInModel, responsável por carregar os dados da View para o Modelo. Além disso, temos também o método BindInView, responsável pela operação inversa, ou seja, carregar os dados do modelo para a View, exibindo os mesmos para o usuário. Por fim temos o método ClearControls responsável por limpar os valores da tela.

A **Listagem 10** segue o mesmo padrão da **Listagem 9**, com o controller responsável pela comunicação entre a View e o Model,

Listagem 8. Implementação dos métodos adicionais BaseController

```
function TBaseController<TFormulario, TModelo>.GetTotalCadastrados: Integer;
begin
 result := FModelosCadastrados.Count;
end;
procedure TBaseController<TFormulario, TModelo>.CloseForm(Sender: TObject;
var Action: TCloseAction);
begin
 Action := TCloseAction.caHide;
 FModelo := nil;
 ClearControls();
 FFormulario.Hide();
end;
procedure TBaseController<TFormulario, TModelo>.SalvarClick(sender: TObject);
begin
 Save();
end;
procedure TBaseController<TFormulario, TModelo>.ShowFormModal(modelo:
TModelo);
begin
 FModelo := modelo;
 FFormulario.ShowModal();
end;
procedure TBaseController<TFormulario, TModelo>.Save;
begin
 BindInModel();
 if(FModelo.Id = 0)then
 begin
   FModelo.Id := GetTotalCadastrados() + 1;
   FModelosCadastrados.Add(FModelo);
   FFormulario.lblTotalCadastrado.Caption := 'Total de Cadastros: ' + IntToStr
   (GetTotalCadastrados());
 end;
end;
end.
```

Listagem 9. Implementação do ClienteController

```
unit UClienteController;
interface
uses UBaseController, UFrmCadastroCliente, UCliente;
type
 TClienteController = class(TBaseController<TFrmCadastroCliente,TCliente>)
   procedure BindInView();override;
   procedure BindInModel();override;
   procedure ClearControls();override;
 end;
implementation
 uses
   System.SysUtils;
   { TClienteController }
 procedure TClienteController.BindInModel;
 begin
   inherited;
   FModelo.Nome := FFormulario.edtNome.Text;
   FModelo.Idade := StrToInt(FFormulario.edtIdade.Text);
   FModelo.Ativo := FFormulario.cbbAtivo.ItemIndex = 0;
 end;
 procedure TClienteController.BindInView;
 begin
   inherited;
   FFormulario.edtNome.Text := FModelo.Nome;
   FFormulario.edtIdade.Text := IntToStr(FModelo.Idade);
   if(FModelo.Ativo) then
     FFormulario.cbbAtivo.ItemIndex := 0
   else
     FFormulario.cbbAtivo.ItemIndex := 1;
   end;
   procedure TClienteController.ClearControls;
   begin
     inherited;
     FFormulario.edtNome.Text := '';
     FFormulario.edtIdade.Text := '';
     FFormulario.cbbAtivo.ItemIndex := 0;
 end;
end.
```

Listagem 10. Implementação do FornecedorController

```
unit UFornecedorController;
interface
uses UBaseController, UFrmCadastroFornecedor, UFornecedor;

type
 TFornecedorController = class(TBaseController<TFrmCadastroFornecedor,
 TFornecedor>)
   procedure BindInView();override;
   procedure BindInModel();override;
   procedure ClearControls();override;
 end;

implementation
 { TFornecedorController }
 procedure TFornecedorController.BindInModel;
 begin
   inherited;
   FModelo.CNPJ := FFormulario.edtCNPJ.Text;
   FModelo.RazaoSocial := FFormulario.edtRazaoSocial.Text;
   FModelo.NomeFantasia := FFormulario.edtFantasia.Text;
 end;
 procedure TFornecedorController.BindInView;
 begin
   inherited;
   FFormulario.edtCNPJ.Text := FModelo.CNPJ;
   FFormulario.edtRazaoSocial.Text := FModelo.RazaoSocial;
   FFormulario.edtFantasia.Text := FModelo.NomeFantasia;
 end;
 procedure TFornecedorController.ClearControls;
 begin
   inherited;
   FFormulario.edtCNPJ.Text := '';
   FFormulario.edtFantasia.Text := '';
   FFormulario.edtRazaoSocial.Text := '';
 end;
end.
```

realizando o carregamento dos dados entre os mesmos com os métodos BindInView e BindInModel, porém desta vez para o cadastro de Fornecedor.

Criando o formulário principal e juntando tudo

Após criados os modelos, as views e os controllers, só falta agora testarmos nossa aplicação criando um formulário principal, que utilizará os controllers concretos para abrirmos nossos formulários.

Adicione um novo formulário ao projeto e nomeie o mesmo como FrmPrincipal. Insira dois botões neste formulário, um com caption "Cadastro de Clientes" e nome btnCadastroClientes e outro com "Cadastro de Fornecedores" e nome btnCadastroFornecedores.

Declare duas variáveis no escopo **public** do formulário, conforme mostrado a seguir:

```
ClienteController:TClienteController;
FornecedorController:TFornecedorController;
```

Agora basta implementarmos o evento click dos botões para instanciar os controllers e então abrir os formulários, como mostrado na **Listagem 11**.

Generics na API do Delphi

Agora que já vimos como implementar nossas próprias classes genéricas, é só colocar a mão na massa para aumentar a abstração e reuso de nossos projetos. Por outro lado, antes de colocar a mão na massa, é interessante conhecer o que o próprio Delphi já nos fornece para não corrermos o risco de reinventar a roda. A seguir na **Tabela 1** segue uma lista dos principais recursos do Delphi que utilizam Generics para que você pode utilizar em seus projetos.

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DevMedia

Classes / Interfaces	Descrição
IComparer<T>	Fornece uma interface para comparação dos tipos "menor que", "maior que" or "igual a"
IEqualityComparer<T>	Fornece uma interface para comparação de igualdade, "igual a" ou "diferente"
TComparer<T>	Classe abstrata para implementação da interface IComparer
TEqualityComparer<T>	Classe abstrata para implementação da interface IEqualityComparer<T>
TDelegatedEqualityComparer<T>	Uma classe de comparação que utiliza delegates para determiner igualdade.
TDelegatedComparer<T>	Uma classe de comparação que usa delegates para fazer a comparação.
TStringComparer	Classe usada para comparação de strings.
TArray	Esta classe não contém dados, mas fornece métodos para trabalhar com arrays como por exemplo os métodos Sort e BinarySearch.
TEnumerator<T>	Implementa o design pattern Iterator para seus dados.
TList<T>	Classe genérica que permite o armazenamento de uma coleção genérica de qualquer tipo de dado, sejam objetos ou tipos primitivos.
TQueue<T>	Classe genérica que fornece uma fila de objetos, com algoritmo Filo(first in / last out)
TStack<T>	Classe genérica que fornece uma pilha de objetos com algoritmo FiFo(first in / first out).
TPair<TKey, TValue>	Estrutura simples de par chave/valor.
TDictionary<TKey, TValue>	Coleção de estruturas chave/valor.
TObjectList<T: class>	Coleção genérica para armazenamento de objetos.
TObjectQueue<T: class>	Fila genérica para objetos.
TObjectStack<T: class>	Pilha genérica de objetos.

Tabela 1. APIs do Delphi que utilizam Generics

Conclusão

Neste artigo vimos o que são tipos genéricos no Delphi, como utilizar os mesmos e as restrições que podemos aplicar a eles. Construímos um exemplo básico com um MVC, com a criação de um controller genérico e outros controllers concretos. Buscamos demonstrar como criar seus tipos genéricos e também quais as classes nativas do Delphi que implementam este recurso.

Escolhi o exemplo do MVC, pois assim conseguimos acrescentar um pouco mais ao leitor, além de mostrar que Generics podem e devem ser explorados juntamente com padrões e boas práticas de desenvolvimento, ao invés de se limitar a classes de serviços específicos.

Os tipos genéricos nos fornecem grande flexibilidade no desenvolvimento. Com eles podemos desacoplar mais nosso código e torná-lo mais elegante. Com o uso de tipos genéricos, conseguimos criar classes mais coesas que dispensam a necessidade de typecasting. Aliando generics a design patterns e boas práticas de desenvolvimento, podemos obter maior reuso e facilitar a manutenção de nosso código.

Use sua imaginação para criar arquiteturas mais flexíveis em seus projetos no dia a dia. A combinação de Generics com Design pattern é explosiva, onde podemos conseguir um aumento de produtividade altamente relevante através do maior reuso de código.

Ricardo Coelho
(rcoelhorj@gmail.com)
Graduado em Análise de Sistemas, Pós-graduando em Engenharia de software, mais de 7 anos de experiência em desenvolvimento de software, onde atuou em projetos de diversos seguimentos como varejo, telecomunicações, ERPs etc e nas mais variadas plataformas(Desktop, Web e Mobile). Articulista do portal devmedia e das revistas ClubeDelphi e Easy .net Magazine. Entusiasta de métodos ágeis, atualmente atua como ScrumMaster em projetos de software voltados para a área de petróleo e gás além de se dedicar ao aprofundamento de seus estudos na área de GIS e Mobile.

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,
assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486

TENHA ACESSO ILIMITADO A TODO CONTEÚDO DO SITE DEVMEDIA

SEJA UM ASSINANTE MVP!

Tenha acesso garantido a qualquer tema que você possa necessitar!

VIDEOS -AULAS

O assinante MVP acessa todas as vídeo-aulas do site DevMedia. Nenhum outro produto dá essa possibilidade. São mais de **7.000 video-aulas.**

CURSOS

Nosso site possui atualmente **+ de 250 cursos online** com a qualidade que você já conhece e confia. Com o MVP você terá acesso a todos eles.

REVISTAS

Assinante MVP acessa todas as nossas revistas na versão digital, incluindo todas as edições anteriores. Quando pintar aquela dúvida você terá mais de **3.500 artigos para consultar!**

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

A assinatura MVP te dá a oportunidade de aumentar seu conhecimento em diversos assuntos, com qualidade e economia.

Invista na sua carreira se torne um profissional ainda mais completo!

POR APENAS R$59,90 POR MÊS

DevMedia

Acesse: www.devmedia.com.br/mvp